VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra kybernetiky a biomedicínského inženýrství

Univerzální databázový systém pro definici technologických objektů v řídicích aplikacích

Universal Database System for Technological Objects Definition in Control Applications

2019 **Bc. Šárka Mikolajková**

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra kybernetiky a biomedicínského inženýrství

Zadání diplomové práce

Student: **Bc. Šárka Mikolajková**

Studijní program: N2649 Elektrotechnika

Studijní obor: 2612T041 Řídicí a informační systémy

Téma: Univerzální databázový systém pro definici technologických objektů v řídicích aplikacích
Universal Database System for Technological Objects Definition in Control Applications

Jazyk vypracování: čeština

Zásady pro vypracování:

Diplomová práce se zabývá analýzou, návrhem a implementací softwarové aplikace, která bude umožňovat programátorům řídicích systémů rozdělení dílčích řízených technologií do jednotlivých technologických objektů, definovat vlastnosti jednotlivých objektů a technologických vazeb mezi nimi. Výsledná aplikace bude založena na translační databázi (Microsoft SQL) v prostředí .NET. Jejím cílem je přenos navržených objektů do různých vývojových prostředí pro programování a konfiguraci řídicích systémů pro širší množinu výrobců. Konkrétní zadání se zabývá realizací rozhraní produktů firmy Siemens a to TIA portál a Step 7.

Body zadání:
1. Rešerše dostupných řešení.
2. Analýza, specifikace požadavků a návrh výsledné aplikace.
3. Návrh a realizace databáze pro podporu cílů práce.
4. Implementace aplikace v prostředí .NET.
5. Test aplikace v reálném prostředí.
6. Závěr a zhodnocení dosažených výsledků.

Seznam doporučené odborné literatury:

[1] CORONEL, Carlos and Steven MORRIS. *Database Systems: Design, Implementation, & Management*. 12 edition. Boston: Course Technology, 2016. ISBN 978-1305627482.
[2] RINALDI, John S. *OPC UA - Unified Architecture: The Everyman's Guide to the Most Important Information Technology in Industrial Automation*. 1 edition. [s.l.]: CreateSpace Independent Publishing Platform, 2016. ISBN 978-1530505111.
[3] KRUTZ, Ronald L. *Industrial Automation and Control Systems Security Principles*. 2 edition. [s.l.]: International Society of Automation, 2016. ISBN 978-1941546826.
[4] MEHTA, B. R. a Y. Jaganmohan REDDY. *Industrial process automation systems: design and implementation.* Amsterdam: Elsevier (Butterworth-Heinemann), 2015. ISBN 978-0128009390.
[5] BENEŠ, Pavel, et al. *Automatizace a automatizační technika. 1, Systémové pojetí automatizace.* Brno: Computer Press, 2012. ISBN 978-80-251-3628-7.

Formální náležitosti a rozsah diplomové práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí diplomové práce: **Ing. Zdeněk Slanina, Ph.D.**

Datum zadání: 01.09.2017

Datum odevzdání: 30.04.2019

doc. Ing. Jiří Koziorek, Ph.D.
vedoucí katedry

prof. Ing. Pavel Brandštetter, CSc.
děkan fakulty

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracovala samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

Datum odevzdání diplomové práce 30. 4. 2019 .....

Bc. Šárka Mikolajková

Poděkování

Touto cestou bych ráda poděkovala vedoucímu diplomové práce Ing. Zděňkovi Slaninovi, Ph.D. za odborné vedení a cenné rady při zpracovávání této práce. Dále firmě Ingeteam a.s, která umožnila vznik této práce a v neposlední řadě mé rodině a přátelům za podporu během celého studia.

Prohlášení firmy Ingeteam a.s.

Souhlasím se zveřejněním této diplomové práce dle požadavků čl. 26, odst. 9 Studijního a zkušebního řádu pro studium v magisterských programech VŠB-TU Ostrava.

V Ostravě 25. 4. 2019

.....

Ing. Jaroslav Tyšer

Abstrakt

Tématem této diplomové práce je návrh a implementace softwarové aplikace pro tvorbu definic technologických objektů pro proces vývoje řídicích programů automatizovaných systémů. Systém je založen na relační databázi, která je uložištěm dat, jejíž vzájemné vazby utváří definici technologického objektu jako takového. Aplikace je vytvořena na platformě .NET Framework a pro práci s databázovými daty je využit Entity Framework. Dále umožnuje komunikaci s vývojovým prostředím TIA Portal pomocí nástroje TIA Portal Openness. S využitím této komunikace je možné na základě logických vazeb v řídícím programu doplňovat do databáze specifická data, která jsou využita pro generování textových podkladů pro vizualizace. Součástí práce je rozbor problematiky definic technologických objektů a užitých prostředků k realizaci celého systému.

Klíčová slova

Entity Framework, PLC, SQL Databáze, Technologické objekty, TIA Portal Openness, Objektově relační mapování, Windows Presentation Foundation

Abstract

Topic of this master thesis deals with design and implementation of software application for creating definitions of technological object that are used for development of control program for automation systems. The system is based on a relational database which represents data repository. Relationships between entities in database makes the definition of technological object. Created application is based on .NET Framework platform and for operation with database data is used Entity Framework. It also allows the communication with development environment TIA Portal due to includition of TIA Portal Openness tool. This communication is used for adding specific data to database which are based on logical links in the control program. These data are used for generating text descriptions for visualization. Part of the work is an analysis of the process of creating technological objects definitions and the description of tools and resources used for implementation of the entire system.

Keywords

Entity Framework, PLC, SQL Database, Technological Objects, TIA Portal Openness, Object Relational Mapping, Windows Presentation Foundation

Obsah

Seznam použitých symbolů a zkratek

API	Application Programming Interface (Rozhraní pro programování aplikací)
Cflt	Critical Fault (Označení pro indikaci kritické chyby skupin)
CRUD	Označení metod pro vytvoření, čtení, úpravu a smazání. (create,read,update,delete)
DLL	Dynamic Link Library (dynamická knihovna)
DTO	Data transfer object (Objekt pro přenos dat)
FBD	Function Block Diagram (Jazyk funkčních bloků)
HMI	Human Machine Interface (Uživatelské rozhraní - vizualizace)
IL	Interlock (Specifická skupina podmínek)
LINQ	Language-Integrated Query (Jazykově integrované dotazování)
MVVM	Model–View–Viewmodel
ORM	Object relational mapping (Objektové relační mapování)
PID	Označení regulátoru složeného z proporcionální, integrační a derivační části
PLC	Programmable Logic Controller (Programovatelný logický automat)
SQL	Structured Query Language (Strukturovaný dotazovací jazyk)
UI	User Interface (Uživatelské rozhraní)
WPF	Windows Presentation Foundation
XAML	Extensible Application Markup Language
XLSM	Přípona souboru vytvořeného tabulkovým editorem Excel s využitím maker
XML	Extensible Markup Language

Seznam obrázků

Úvod

Tato diplomová práce se zabývá návrhem a implementací softwarové aplikace pro tvorbu definic technologických objektů. Pojem technologický objekt je označení pro veškeré prvky, které se v dané řízené technologii vyskytují. Vytvoření takzvané definiční listiny, která tvoří seznam všech objektů v dané technologii, je prvním krokem při zahájení prací na novém projektu, protože představuje podklady pro tvorbu řídicího programu. Definované objekty pak odpovídají jednotlivým funkčním blokům ve vývojovém prostředí pro řídicí program PLC. Cílem práce je tedy vytvořit systém, který umožní programátorům tvorbu definic technologických objektů, jejich zařazení podle logických a technologických vazeb a specifikaci vlastností. Dále tento software umožní přenos vytvořených objektů do vývojového prostředí, kde jsou programátorem doplněny logické vazby mezi objekty a jejich proměnnými, které tvoří řídicí program. Na základě těchto vazeb je pak možné vytvořenou aplikací generovat podklady pro další části automatizovaného systému, například pro tvorbu vizualizací řízeného procesu. Vznik popisovaného nástroje přináší zefektivnění procesu vývoje a tvoří základ pro unifikaci přístupu k vývoji řídicích programů PLC napříč různými výrobci. Tato práce se zabývá realizací pro produkty firmy Siemens a byla vytvořena ve spolupráci s firmou Ingeteam a.s., která působí v oblasti automatizace technologických procesů.

Navrhovaný systém je založen na SQL databázi a obslužné aplikaci vytvořené na platformě .NET Framework, která umožnuje správu databázových dat a jejich transformaci do vhodné podoby pro prezentaci uživateli. Dále pak na komunikaci s programovacím prostředím TIA Portal pomocí nástroje TIA Portal Openness.

Úvodní kapitoly práce se zabývají rozborem a analýzou vývoje řídicího programu firmou Ingeteam a.s. a podobou současné definiční listiny. Následuje shrnutí požadavků na vyvíjený systém, které byly stanoveny na základě využívaného standardu. Zde je také uveden celkový koncept univerzálního systému, jehož centrální prvek tvoří vytvořená aplikace. Dále následuje část věnována popisu struktury databáze pro definice technologických objektů. Struktura je navržena tak, aby bylo možné dodržovat zažitá pravidla při vývoji řídicích programů. V kapitolách jsou popsány dílčí části databáze, spolu s vysvětlením vazeb mezi jednotlivými entitami.

K samotné realizaci práce bylo využito několik realizačních prostředků, které jsou v konkrétních souvislostech popsány. Jedná se o principy pro výstavbu uživatelského rozhraní na základě subsystému Microsoft Windows Presentation Foundation, postupy vytvoření databáze a využití prostředků pro práci s jejími daty skrze Entity Framework, popisy užitých návrhových vzorů a v neposlední řadě popis využití produktu TIA Portal Openness. Pro využití funkcionality nabízené tímto nástrojem byly vytvořeny třídy obalující funkce DLL knihoven *Siemens.Engieneering.dll*, jenž jsou součástí instalace tohoto produktu.

Kapitola zabývající se realizací aplikace obsahuje části věnující se vnitřní struktuře softwaru a architektuře pro práci s daty. Součástí je popis uživatelského rozhraní a popis principů spojených s transformací databázových dat. Jedná se o jejich zpracování, prezentací uživateli, modifikace a převod zpět do podoby určené pro uchování nesené informace v databázi. Dále jsou popsány automatizované procesy uživatelského rozhraní a mechanismy pro práci s textovými částmi definic. Průběh vývoje a testovaní systému je shrnut v závěru práce. Součástí je také zhodnocení výsledků spolu s podměty k dalšímu rozvoji systému.

1. Popis využívaného standardu pro programování řídicí technologie

Přínos zavedení automatizační techniky do řídicích procesů obecně přináší zvýšení nejen funkční efektivity, ale také například zvýšení bezpečnosti celého provozu a v neposlední řadě úspory z hlediska časové a finanční náročnosti provozu.

Pro řízení technologických procesů téměř ve všech oblastech průmyslu jsou hojně využívány programovatelné logické automaty (PLC). Jsou neodmyslitelnou součástí řízení automatizovaných procesu výrobních a obráběcích linek, těžebního průmyslu, hutí, distribučních a manipulačních systémů továren, překladišť nebo přístavů. PLC pracují na principu zpracovávání signálů a informací, jejichž zdrojem je daný technologický proces. Tato data jsou pomocí naprogramovaných sekvenčních logických obvodů a časových funkcí zpracovávány na výstupní řídicí signály, které jsou nutné pro ovládání jednotlivých prvků daného procesu. Hlavní část programu je psána v jazyce využívající funkční bloky (FBD) a některé specifické funkce jsou definovány pomocí jazyka využívající strukturovaný text (STL). [1][7][20]

Rozsáhlost a komplexnost projektů současné průmyslové automatizace si žádá unifikaci a standardizaci přístupu k vývoji řídicího software, jejíž součástí je hierarchická architektura při propojování jednotlivých částí do větších celků. Z tohoto důvodu je tato úvodní kapitola věnována analýze a popisu současně využívaného standardu firmou Ingeteam a.s. pro vývoj řídicího software v prostředí TIA Portal (celým názvem Totally Integrated Automation Portal), a Step7.

1.1 Základní vnitřní stavba programu PLC

Typickými zástupci z řad PLC jsou například zařízení Simatic řady S7, výrobce Siemens. Řídicí software programových automatů je složen z *operačního systému* a *uživatelského programu*. *Operační systém* je interní součástí zařízení a řídí všechny funkce a sekvence, které nejsou propojeny s řídicím programem technologie (např. zpracování restartu, volání uživatelského programu, zpracování chyb, práce s pamětí atd.). Tento operační systém je vyvíjen výrobcem pro konkrétní typ zařízení. *Uživatelský program* obsahuje veškeré bloky potřebné k zvládnutí automatizační úlohy. Uživatelský program je složený jednotlivých programových bloků a je do zařízení vložen. Tento program je vytvářen pomocí programovacích rozhraní Step7 nebo TIA Portal. Rozlišujeme čtyři základní typy stavebních bloků programu. Jejich užití přináší možnost strukturalizace a zvýšení celkové přehlednosti programu. To vede ke snadnější údržbě, analýze kódu, detekci možných chybových stavů a jejich odstranění. Strukturalizace také poskytuje možnost tvorby větších univerzálních funkčních celků, které mohou být opakovatelně použity s odlišnými parametry v jiných částech programu nebo zcela novém projektu. [23][24]

Typu bloků:

- Organizační bloky OB – Jedná se o bloky, které jsou volány operačním systémem a tvoří tak rozhraní mezi operačním systémem kontroléru a uživatelským programem. Mohou mít na starost na příklad chování při spouštění zařízení, řízení cyklického zpracování procesů, přerušení řídicích procesu nebo ošetření chybových stavů. Hlavní organizační blok *Main*, který je pokaždé cyklicky zařízením vyvolán, je vytvářen automaticky jako základ řídicího programu.

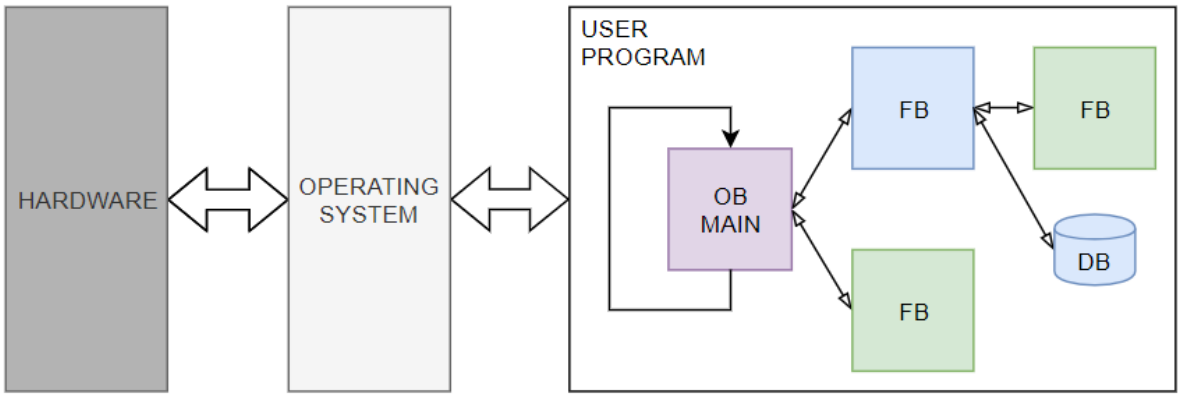
- Funkce FC – Představují logické bloky, kterým není vyhrazen konkrétní prostor v paměti (nemají přiřazen příslušný datový blok). Hodnoty proměnných uvnitř bloku jsou uchovávány ve vyrovnávací paměti pouze v době vykonávání funkce, po jejím ukončení jsou vymazány.

- Funkční bloky FB – Tyto bloky představují funkce nebo sekvence logických funkcí, jejichž hodnoty jsou ukládány v paměti skrze přiřazený datový blok.

- Datové bloky DB – Tyto bloky představují deklaraci prostoru v paměti pro ukládání uživatelských dat a výstupů z jednotlivých bloků. Logické instrukce nejsou v těchto blocích obsaženy. Jsou rozlišovány dva typy datových bloků. Prvním zástupcem jsou *Globální* datové bloky, které mohou být využívány všemi kódovými bloky projektu. *Lokální* (*Instanční*) datové bloky jsou přiřazeny specifickým funkčním blokům. Jejich struktura odpovídá rozhraní bloku FB a mohou být použitý pouze konkrétním blokem.

[9][24]

Na níže uvedeném Obr. 1, je znázorněno blokové schéma zachycující uspořádání řídicího kontroléru.



Obr. 1 Blokové schéma uspořádání programového řízení technologie

1.2 Definiční listina

Prvním krokem při zahájení prací na novém projektu je vytvoření takzvané definiční listiny. Jedná se o seznam všech objektů vyskytujících se v projektu.

Celá definiční listina a její části jsou v současné době realizovány pomocí šablony vytvořené v programu Excel. Při zakládání nového projektu se většinou vychází z definic využitých v jiném projektu, která je následně upravována pro současný projekt. Tento koncept byl vytvořen na základě mnohaletých zkušeností s vývojem řídicích technologií a postupným zdokonalováním byly soubory doplněny o makra, která práci programátorům značně usnadňují. Listina je vytvořena na základě technologického schématu (tzv. flowsheet). Jedná se o dokumentaci řízené technologie, která zachycuje

veškeré její prvky a vzájemné funkční vazby mezi nimi. Pojem řízená technologie je komplexní označení pro řízený proces, včetně všech jeho součástí (překladiště, navíječky, čerpací stanice, apod.)

Objekty jsou děleny na:

- Technologické (čerpadlo, pohon, manipulační rameno, dopravník)
- Elektrické (měnič, měření el. veličin)
- Logické (automatické ovládací sekvence, PID regulátor, shareDb, comDb)

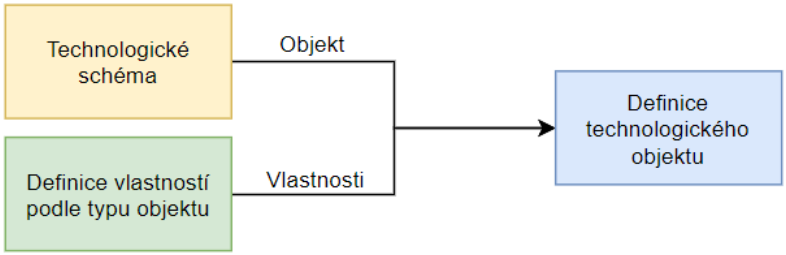
Těmito *objekty* jsou části řídicí technologie jako ventily, motory, frekvenční měniče, ale také například měření teplot či rychlosti. Řadíme zde i objekty, které nejsou přímou součástí technologie. Jedná se o objekty softwarové a elektrické. Například sdílené a komunikační datové bloky (označovány jako ShareDb a ComDb), diagnostiky nebo PID regulátory.

Každý objekt v listině je definován svým názvem, který je totožný s názvem v dokumentaci. Dále popisem představujícím detailnější popis zařízení, přiřazeným typem, případně dalšími specifickými vlastnostmi.

V prostředí Step7 nebo TIAPortal je objekt reprezentován svým instančním datovým blokem stejného názvu. Jak již bylo zmíněno datový blok je vymezené místo v paměti PLC, ve kterém jsou uložena data příslušného objektu, aktuální stavy a nastavené parametry.

Objekty mohou být dále rozřazeny do skupin. *Skupina* sdružuje objekty, které tvoří dílčí části technologického celku a jsou určeny opět na základě technické dokumentace a strukturo celého řízeného systému. Skupina obsahuje objekty, ale může se skládat i z dalších skupin sdružující další objekty. Příkladem skupiny může být část výrobní linky zajišťující vykládku materiálu. Objekty této skupiny jsou například dopravníky a separátory, které se dále skládají z vlastních objektů, kterými mohou být pohony a měniče. Tyto skupiny jsou také objektem definiční listiny. Pro každé PLC zařízení je vytvořena tato definiční listina, která napomáhá programátorům v orientaci v dané technologii. Jasně tak definuje prvky, se kterými pracuje při práci na konkrétní části řídicí technologie. Toto větvení od vyšších celků k menším je praktikováno do adekvátní míry.

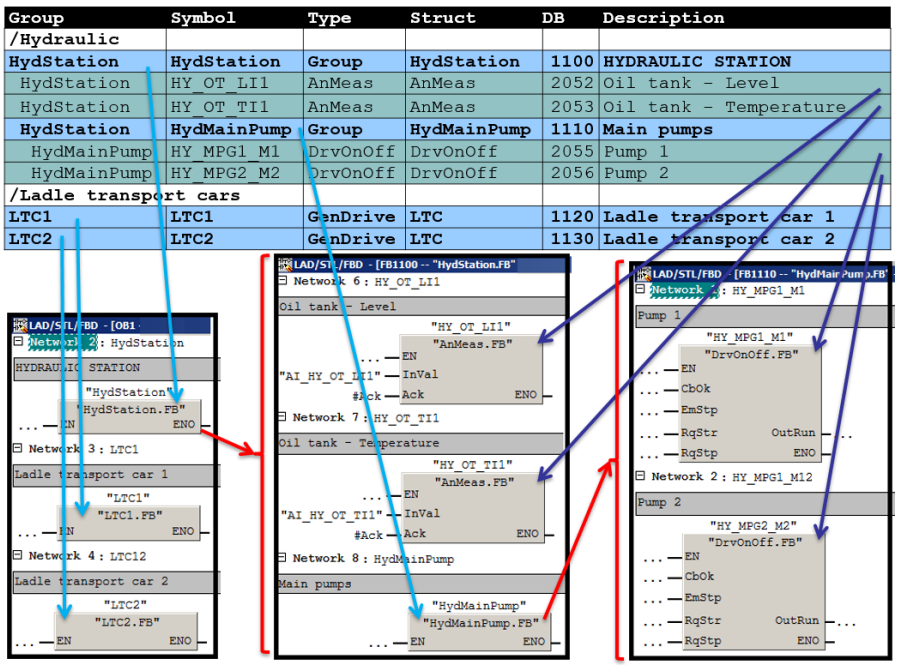
V definici jsou rozlišovány dva typy skupin nesoucí označení *GenDrive* a *Group*. Jedná se o rozlišení skupin podle toho, zda objekty, které ji náleží, představují konkrétní zařízení nebo kus technologie. Dílčí celek označován jako *Group* představuje technologickou skupinu například okruh hydrauliky, chlazení nebo mazání. Skupina *GenDrive* sdružuje objekty, které představují typicky nějaké zařízení a složitější celky, například dopravník. Z pohledu uživatele resp. operátora, se tato skupina jeví stejně jako například nějaké jednoduché zařízení, které má pouze základní stavy on/off. Avšak z pohledu programátora má tento objekt ještě další pod-objekty, které spolu pracují a souvisí. *GenDrive* se od skupiny typu *Group* z pohledu definice ani programování nijak neliší, toto dělení je zavedeno jen pro specifičtější roztřízení objektů.



Obr. 2 Komponenty pro definici objektu

Každá instance definiční listiny je mimo jiné definována svým Typem. Definice jako taková vychází ze dvou částí. Schéma komponent pro definici objektů je uvedeno na Obr. 2. První část je dána technickým řešením daného projektu, čili samotnou existencí objektu jako takového. Druhou část představuje soubor přirazených vlastností, které jsou definovány právě jeho typem. *Typ* představuje v zásadě předpis pro daný objekt, lze si jej představit jako definici třídy v objektově orientovaných programovacích jazycích. V daném projektu tak může být několik objektů, které jsou instancemi stejného typu. Názorným vysvětlením může být opět konkrétní příklad. Uvažujme, že daným objektem technologie je měření teploty. Teplota je velmi častým parametrem měřeným na mnoha místech systému. Nadefinováním typu pro digitální měření (v technologii označován jako *DgMeas*) lze všechna tato měření zpracovávat pomocí jednoho funkčního bloku. V tomto případě je blok *DgMeas* v programu volán pokaždé s jinými vstupy a jsou tak zpracovávaná konkrétní data odpovídající příslušné instancí. Některé typy se mohou v různých projektech opakovat nebo se jen nepatrně lišit a mohou být v programu zkopírovány z předešlých projektů. Jiné mohou být unikátní pro daný případ. Definice typů jsou vždy jednotné napříč všemi PLC v rámci celého projektu a v mnoha případech vycházejí z předcházejících projektů, které se upravují podle konkrétních potřeb. Pro objekty typu skupin pak rozlišujeme výše zmiňované typy *Group* a *GenDrive*.

Na uvedeném Obr. 3 je uvedena ukázka části definice technologických objektů. V tabulce je zaznamenána definice objektů spadajících do hlavní nadřazené skupiny hydraulické stanice - *HydStation*. Z tabulky je zřejmé, že daná skupina obsahuje objekty analogového měření *HY_OT_LI1* a *HY_OT_TL1*. Jedná se o objekty o analogového měření, což je charakterizováno zvoleným typem *AnMeas*. V této skupině objektů je také definována podskupina *HydMainPump*, která obsahuje další vlastní objekty.



Obr. 3 Ukázka části definice objektů a jejich hierarchie, spolu se znázorněním odpovídajících bloků [20]

1.3 Struktury proměnných instančních datových bloků

Z předcházejících částí je zřejmé, že objektu uvedenému v definiční listině náleží příslušný datový blok. Obsah datového bloku je dán přiřazeným typem danému objektu. Typ tedy specifikuje, jaké proměnné danému objektu, resp. bloku náleží. Tyto proměnné jsou podle svého účelu rozdělovány do skupin, a tím definují vnitřní strukturu bloku.

Typicky se jedná o struktury:

- SP (Setpoint) – zadané hodnoty a parametry
- CMD (Command) – povely z HMI stanice
- STS (Status) – aktuální stav zařízení
- FLT (Fault) – aktuální poruchy a varování

Toto rozdělení je pak využíváno dalšími částmi programu a vizualizaci jako interface pro přístup k proměnným datových bloků. K proměnným je přistupováno pomocí specifických odkazů neboli tzv. tagů. Tag může vypadat následovně: *Hydraulic_Sts_Run*. Tento konkrétní tag nese bitovou informaci o tom, zda je zařízení v chodu (ano/ne). *Hydrauclic* je název objektu s příslušného datového bloku a *Sts* představuje strukturu dat uvnitř bloku, kde je konkrétní proměnná *Run* uložena. Často nastává situace, že různé typy objektů mohou mít stejný předpis některého z těchto rozhraní pro vizualizaci. V praxi to znamená, že například různé typy ventilů, které se liší stavbou, vnitřními objekty, ovládáním nebo řídicím kódem, se z pohledu vizualizace jeví jako stejné objekty – ventil. Pracuje s nimi stejně, protože pro operátora jsou v obou případech podstatné stejné informace (například stav on/off).

Tento přístup je umožněn právě díky existenci definic struktur, které odkazují na přesné umístění konkrétní struktury proměnných nesoucích požadovanou informaci.

Speciální skupinou proměnných, které odráží stav objektu, jsou takzvané *Interlocky*. Jedná se o hodnoty odrážející stavy zařízení, které jsou prezentovány skrze HMI. Jejich hodnoty jsou výstupem logických podmínek mezi proměnnými napříč objekty. *Interlock* tedy definuje podmínky pro jednotlivé akce, stavy a jejich závislosti daného funkčního bloku technologie. Jedná se o podmínky, které musí byt splněny, aby objekt mohl přejít z jednoho stavu do druhého. V případě jednoduchého zařízení jako je například motor lze předpokládat, že budou definovány tři základní interlocky pro start (START), běh (RUN) a zastavení motoru (STOP). Ty obsahují podmínky, které musí byt splněny, aby bylo možné akci provést. Pro start motoru bude tedy nutné, aby byla splněna podmínka aktuálních nulových otáček a sepnutých bezpečnostních prvků. Při splnění obou těchto podmínek bude interlock START vyhodnocen jako logická pravda a bude umožněno spuštění zařízení. Aby byl stav motoru vyhodnocen jako RUN, bude předpokládáno splnění podmínky nenulových otáček motoru atd. Při Speciálním případem interloku jsou podmínky pro kritické chyby Cflt (Critical Fault). Z toho vyplývá, že pokud je definována podmínka Cflt je možno danou akci vykonat v případě, že je jeho logická hodnota rovna 0.

Pro často opakující se skupiny proměnných, mohou byt vytvořeny substruktury proměnných, které tyto proměnné sdružují. Nejčastěji se jedná o základní společné části interface pro určité struktury, např. proměnné pro *Status*. Tyto položky jsou pro dané objekty stejné, a proto se v seznamu proměnných pak nachází odkaz na tuto skupinu předpisů proměnných a pak následují speciální položky pro daný typ objektu. Definice společných typů proměnných přináší další zapouzdření přístupu k vývoji technologie, jehož hlavní přínos se projevuje v okamžiku, kdy je nutné provést nějakou obecnou změnu interface. Změny jsou provedeny pouze na jednom místě, v definici substruktury, a promítnou se do všech bloků.

Vnitřní uspořádání proměnných uvnitř datových bloků je uvedeno v příloze II.

2. Účel navrhovaného systému a specifikace požadavků

Cílem této kapitoly je rozbor problematiky zabývající se požadavky na systém, jehož vývoj je předmětem této diplomové práce. První část se zabývá analýzou funkčních požadavků, které s vymezují požadované funkce vyvíjené aplikace, dále následuje shrnutí faktů, které blíže specifikují podmínky pro vývoj. Poslední částí je popis návrhu systému vývoje řídicích programů, jehož součástí by vyvíjená aplikace ve své finální fázi měla být. Shrnuje tak hlavní myšlenku unifikovaného přístupu k řídicím technologiím.

Ze zadání práce vyplývá že, hlavní částí je implementace softwarové aplikace umožňující práci s objekty řídicí technologie, jejich definicemi a strukturami sdružujícími proměnné datových bloku odpovídajících objektů. Definované objekty tvoří základní kostru pro tvorbu programu pro PLC. Cílem je tedy nahradit stávající systém definic technologických objektů pomocí šablony a maker v XLSM souborech (Excel), který byl nastíněn v předcházející kapitole. Motivací k vytvoření takového nástroje je předpokládaný přínos v oblasti v časové úspory a unifikace způsobu definování programových objektů a jejich vnitřních struktur. Tento přístup s sebou přináší zvýšení spolehlivosti z důvodu eliminace lidských chyb například při ručním přepisování.

Jak již bylo zmíněno, práce vzniká ve spolupráci s firmou Ingeteam a.s. a na základě diskuse byly stanoveny následující základní požadavky na vyvíjenou aplikaci nesoucí pracovní název Triton.

2.1 Funkční požadavky

Jádrem systému bude vytvořená relační databáze, ve které budou ukládána data, která jsou nositeli informace o definicích daných objektů. Struktura této databáze (tzn. tabulky, jejich sloupce a vzájemné relace) bude vytvořena na bázi vztahů vyplývajících ze standardu struktury vývoje a stavby PLC programů. To vyžaduje vytvoření vhodného uživatelského rozhraní, skrze které bude navržená databáze spravována. Kromě správy připojené databáze a možnosti připojení k jiné, již existující, databázi bude možné pomocí aplikace při založení nového projektu vytvořit zcela novou databázi, jejíž vybrané tabulky budou automaticky naplněny výchozími daty. Jedná se o neměnná data specifikující například seznamy datových typů, jednotek, kategorie objektů a podobně.

Vytvořená aplikace umožní programátorovi vytvářet definice typů, definice struktur rozhraní, definice proměnných a jejich substruktur. Tyto prvky budou pak použity pro definování již konkrétních objektů dané řídicí technologie, které mohou být zpětně editovány a kopírovány. To umožňuje práci i s již dříve vytvořenými objekty.

Definování struktur zahrnuje:

Zařazení struktury podle typu, který napomáhá třízení struktur podle logických vazeb a definování proměnných dané struktury. Tyto proměnné jsou charakterizovány svým datovým typem (na jehož základě je stanovena adresa v paměti PLC), pozicí, jednotkou a obecným popisem obsahující textové symboly (@ObjText), které představují části textu, které budou při užití dané struktury v konkrétním objektu nahrazeny definovanými texty.

Definování objektu zahrnuje:

Zařazení objektu podle typu, který charakterizuje objekt technologie, zařazení objektu do skupiny (pokud není sám nadřazenou skupinou), vložení popisu objektu, adresy, přiřazení typu sdružující

struktury, které svou vlastní definicí určují interface pro vizualizaci a další části programu. Dále pak definice textů pro náhradu symbolů v obecném popisu struktur a případné definování specifických jednotek.

Součástí uživatelského rozhraní bude také přehled hierarchie a vzájemné návaznosti objektů, jejich skupin a struktur v aktuálním projektu.

Při vytváření definic je nutné zahrnout specifické restrikce, které vyplývají ze struktury PLC programu a jsou spojeny s funkčními předpoklady daného objektu. To spočívá ve filtraci jednotlivých nabídek parametrů, datových typů a volitelných prvků v uživatelském rozhraní.

Během editace a vytváření objektů bude možné měnit pořadí struktur v objektu, na jehož základě budou automaticky určovány velikosti struktur a relativní adresy v paměti PLC vymezené příslušnému datovému bloku.

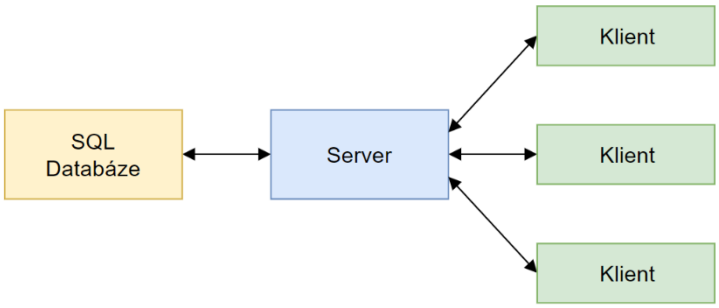
Je nutné zajistit provázanost změn všech vzájemně závislých prvků v systému tak, aby změna některého z parametrů daného objektu, jeho typu nebo struktury byla promítnuta i na jeho referencované položky. Stejně tak změny provedené v definicích typů nebo struktur musí být provázány na objekty, jejichž definice se změny týkají.

Další funkcí vyvíjené aplikace bude přenos informací mezi databází a vývojovým prostředím pro tvorbu PLC programu. Aplikace bude umožnovat na základě logických vazeb mezi objekty, vytvořených programátorem ve vývojovém prostředí, generovat položky jednotlivých interlocků a podklady pro vizualizace. Mezi tyto podklady patří textové popisy jednotlivých položek interloků a jejich překlady. K tomuto procesu budou využity exportované datové bloky z TIA Poratlu, pomocí vestavěné nadstavby TIA Portal Openness.

2.1.1 Požadovaná topologie systému

S přihlédnutím na předpokládané využití aplikace v prostředí firemního oddělení automatizace, kde probíhají práce na vývoji řídicích technologií více programátory najednou a nejedná se tedy o jedno lokální pracoviště, je nutné zahrnout i požadavky vyplývající z této situace.

Jednou z možností je rozdělení systému na nezávislou klientskou část, která bude komunikovat s oddělenou serverovou částí. Tato serverová část bude zajišťovat spojení s databází. Topologie je schematicky znázorněna na Obr. 4. Toto uspořádání umožní práce nad jednou databází více uživatelům z více míst právě pomocí klientských aplikací umístěných na pracovištích programátorů. Předpokládaným místem, kde bude serverová část aplikace umístěna a spuštěna je v místě úložiště databáze.



Obr. 4 Topologie vyvíjené aplikace

2.2 Omezení a nefunkční požadavky

V této podkapitole budou nastíněny aspekty, které blíže specifikují vlastnosti vyvíjené aplikace a proces jejího vývoje, avšak nemají vliv na funkcionalitu, kterou má plnit. Dále budou uvedena omezení plynoucí z problematiky týkající se univerzálnosti systému.

Prvním pomyslným bodem jsou požadavky v oblasti využitých softwarových platforem. Aplikace bude vyvíjena na platformě .NET a určena pro desktopová zařízení využívající Microsoft Windows a grafické rozhraní Microsoft Windows Presentation Foundation (WPF). Databáze pro ukládání dat bude založena na struktuře MySQL z čehož vyplývá využití strukturovaného dotazovacího jazyka SQL. Dalším požadavkem je sestavovat zdrojový kód aplikace tak, aby byla dodržena struktura a modulárnost architektury aplikace umožňující případné úpravy a rozšiřitelnost.

Výkonnost aplikace v oblasti rychlosti komunikace s databází a zpracování jejich dat není nijak zvlášť kritická, protože se nejedná se o žádnou řídicí aplikaci, ale o editační nástroj. Proto nejsou na rychlost odezvy kladeny žádné výjimečné požadavky. Je však nutné zajistit takovou výkonnost aplikace, aby uživatele při práci neomezovala.

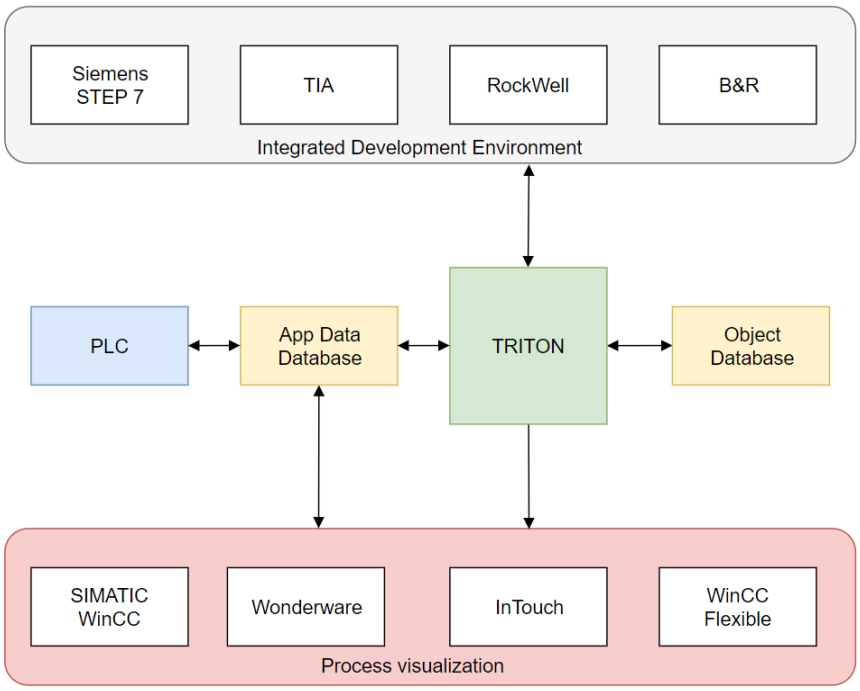
2.3 Definice postupu prací

Na základě výše určených požadavků na vyvíjený systém byly stanoveny hlavní kroky vývoje a implementace systému. Práce budou zahájeny vytvořením struktury SQL databáze na základě modelu objektů a struktur PLC Programu. Následujícím krokem bude vytvoření uživatelského rozhraní pro komunikaci s databází a pro zapracování technologických požadavků na vytváření a práci s objekty. Zde spadá návrh a implementace metod zajišťující zpracování dat z databáze, jejich interpretaci uživateli, automatické generování objektů a zajištění provázanosti dat. Další fází bude návrh přenosu vytvořených objektů mezi databází a programovacím prostředím. Při návrhu systému a metod musí být přihlíženo na budoucí požadavek univerzálnosti systému napříč výrobci, proto je nutné, aby bylo možné části kódu v budoucnu snadno modifikovat.

Toto rozvržení prací, kdy na sebe jednotlivé kroky chronologicky navazují, by mělo poskytovat prostor pro zahrnutí případných změn požadavků. Reálně lze předpokládat, že během vytváření daného systému budou vznikat nové skutečnosti, které budou vyžadovat úpravy v předcházejících krocích.

2.4 Začlenění aplikace do systému vývoje řídicí technologie

Vytvoření výše popisované aplikace tak přináší celkové zefektivnění a unifikaci přístupu k vývoji řídicích programů PLC. Aplikace tvoří základ vývojového nástroje, který by měl ve své konečné fázi tvořit rozhraní mezi jednotlivými procesy, které jsou nutné k vývoji řídicí technologie využívající PLC. Začlenění aplikace do vývoje je zobrazeno na níže uvedeném funkčním schématu na Obr. 5, které bude podrobněji rozebráno a vysvětleno.



Obr. 5 Schéma navrhovaného procesu vývoje řídicího programu

Schéma představuje návrh podoby a funkce univerzálního vývojového systému napříč platformami různých výrobců. Řídicí technologie se jako celek skládá z programově obsluhované části řídící daný proces, vizualizační části, databáze parametrů PLC a samotného řídicího PLC.

První pomyslnou vrstvu systému uvedeného na blokovém schématu tak tvoří prostředky poskytující programovací prostředí pro vývoj programu pro PLC. Zde je uvedeno několik zástupců, lišících se podle výrobce PLC (Siemens, RockWell, B&R a jiné). Při klasickém vývoji je v této části vytvářen celý řídicí program, značné zvýšení výkonnosti vývoje přináší definiční listina, která poskytuje přehled o daných objektech technologie a jejich vlastnostech. Pomocí navrhované aplikace lze tento proces ještě zefektivnit.

Centrálním blokem procesu je aplikace nesoucí označení Triton, kde jsou uživatelem definovány jednotlivé objekty programu skrze data uložená v databázi. Vytvoření základu pro tento centrální blok je předmětem této diplomové práce. Vytvořené definice objektů jsou uloženy do databáze a importovány do vývojového prostředí pro programování PLC, kde jsou na základě definic vytvořeny objekty tvořící kostru programu. Vytvořené objekty mohou být dále použity jako základ pro jiné projekty, neboť základ řídicích technologií mívá v mnoha případech jen nepatrné rozdíly.

V daném prostředí jsou pak programátorem vytvořeny logické vazby mezi objekty, které tvoří funkční část programu. Aplikace dále umožnuje generování interlockových podmínek pro jednotlivé funkční bloky. Po vytvoření programové části jsou exportovány z vývojového prostředí soubory nesoucí informace o jednotlivých funkčních blocích programu a jeho struktuře ve formě XML souborů. Tyto soubory jsou v aplikaci parsovány a data jsou využita pro generování již konkrétních položek interlocků, které jsou uloženy do databáze.

Další část funkčního schématu představuje tvorbu podkladů pro HMI. Aby bylo možné vyčítat hodnoty a aktuální stavy z PLC pro vizualizaci procesu, je nutné definovat objekty, které jsou nositeli informace o umístění a identifikaci dané proměnné. Jedná se o seznamy tagů, alarmů a trendů. List trendů představuje seznam proměnných (resp. jejich adres), jejichž hodnoty jsou v periodických intervalech vyčítány a ukládány. Lze je tak využít pro trendové sledování. Dále mohou být předávány i odkazy na data na celých struktur. Pomocí nástroje Triton lze také generovat textové popisy jednotlivých tagů, včetně jejich překladů do jiných jazyků.

Třetí pomyslnou část celého systému představuje databáze AppData, do níž jsou ukládány parametry PLC pro řízení procesu. Jedná se o konkrétní proměnné představující například limitní hodnoty teplot a tlaků. Z této databáze pak mohou PLC a HMI vyčítat konkrétní proměnné, například pro tzv. proces *forcování*.

Protože se jedná o velmi komplexní systém, je tato diplomová práce zaměřena na části, které úzce souvisí s problematikou vytváření definic objektů. Jedná se tedy o návrh databáze, komunikaci a práci s databázovými daty, vytvoření uživatelského rozhraní pro možnosti vytváření definic objektů a komunikaci s vývojovým prostředím TIA Portal v potřebném rozsahu pro export funkčních bloků.

3. Struktura databáze pro definice objektů

Jedním z předních kroků při zahájení vypracovávání diplomové práce byl návrh a vytvoření databáze pro navrhovaný systém. Platformou pro databázi je systém Microsoft SQL, samotné vytvoření databáze proběhlo pomocí tzv. *Code First* přístupu s využitím *.NET Entity Frameworku*. Jedná se o postup, kdy je vytvoření databázové struktury podmíněno definováním modelu pomocí tříd v obslužné aplikaci. Bližším popisem softwarových nástrojů a vysvětlením postupů pro vytvoření databáze se zabývá níže uvedená podkapitola 4.3 Entity Framework. Tato kapitola je věnována popisu jednotlivých tabulek, jejich významu, vzájemných vazeb a vysvětlení zbývajících pojmů, které jsou s definicemi technologických objektů a struktur spojeny.

Struktura databáze je podřízena užití využívaní zažitých pravidel při vývoji řídicích programů firmou Ingeteam a.s. Proto se v databázi vyskytují i části, které nejsou v této konkrétní aplikaci dosud využity, ale jsou připraveny pro další rozvoj tohoto složitého systému (bus, node apod.). Základní architektura navržené databáze tak vychází z využívaného standardu, který, je již využíván v jiných aplikačních oblastech a vychází z vlastností objektů a struktur PLC Programu. Struktura byla rozšířena o tabulky a sloupce nutné k umožnění požadované funkcionality konkrétního navrhovaného systému.

Vytvořená databáze jako taková, je nositelem a uložištěm informací o všech prvcích řízené technologie, resp. celého řídicího procesu. Obsahuje definice nejen objektů dané technologie, jež jsou řízeny, ale také další elementy, které jsou potřebné v rámci celého projektu. Jedná se o instance řídicích PLC, definice vygenerovaných interlocků na základě již vytvořeného programu, seznamy vstupů a výstupů, seznamy komunikačních rozhraní, seznamy užitých vizualizací v projektu a v neposlední řadě seznamy textů využívaných právě pro vizualizace včetně jejich překladů. Pomocí vzájemných vazeb lze udržet strukturu technologie a související informace celého řízení na jednom místě.

Jednotlivé entity databáze, představují elementy, pomocí kterých jsou sestaveny definice konkrétních objektů technologie. Sloupce příslušné tabulky, atributy, pak udávají specifické vlastnosti konkrétní položky. Definice je tedy sestavena z několika položek, které představují odkazy na data v různých tabulkách.

Databáze jako taková je tvořena 34 tabulkami, které jsou skrze cizí klíče vzájemně provázány. Primárním klíčem všech tabulek je atribut *Id* představující unikátní identifikační číslo záznamu a ve většině případů je automaticky generováno databází. Této vlastnosti docílíme nastavením parametru *Identity* pro daný sloupec. Výjimkou jsou tabulky defaultních dat, které jsou plněna při vytváření databáze a mají své primární klíče již nastaveny. Kromě specifických sloupců udávající vlastnosti a podobu objektu, obsahují všechny tabulky automaticky generované sloupce vztahující se k historii jednotlivých záznamů. Jedná se o čas a datum vytvoření a modifikace záznamu a jeho verze.

Ve struktuře lze rozlišit tři typy tabulek. Prvním typem jsou tabulky nesoucí informace o samotné definici objektu nebo struktury a tabulky související s celým projektem. Dále se pak jedná o pomocné tabulky pro eliminaci oboustranných vícenásobných vazeb a v neposlední řadě tabulky, které představují typové číselníky specifikující vlastnosti definovaného objektu (např. typ PLC, jazyk projektu, datové typy apod.). Diagram celé databáze je uveden v příloze IV.

3.1 Popis dílčích částí databáze a vzájemných vazeb

V této podkapitole budou popásány tabulky databáze a jejich vztahy tvořící strukturu databáze, která je poměrně rozsáhlá. Orientaci v celém systému usnadňuje fakt, že databázovou strukturu, lze rozdělit do tří dílčích celků, které spolu logicky a funkčně souvisí. Avšak i mezi entitami těch celků existují vzájemné vazby. Stěžejní část databáze je zaměřena právě na definice technologických objektů, další část sdružuje informace vztahující se k celému projektu jako celku a poslední pomyslný úsek je věnován definicím prvků, které souvisí již s vytvořeným řídicím programem.

3.1.1 Vlastnosti projektu

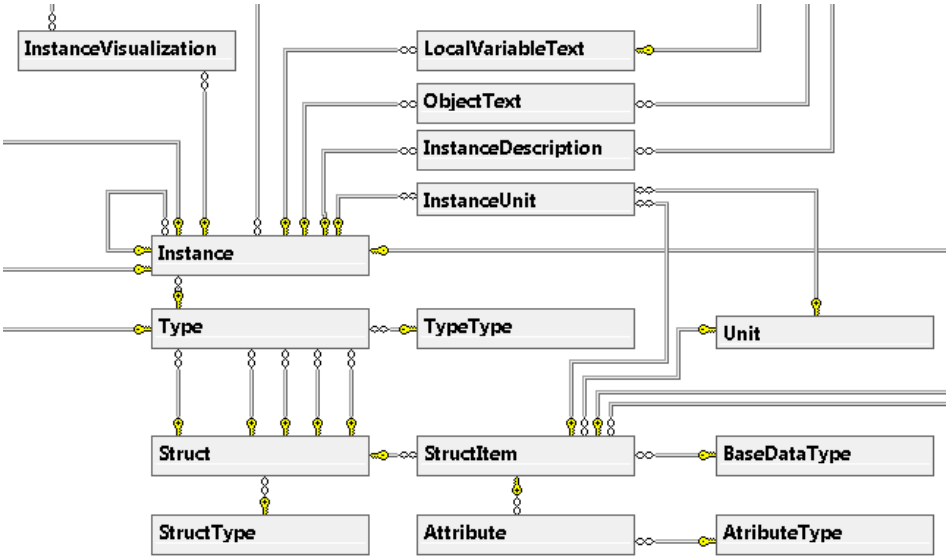
Jak již bylo výše zmíněno, databáze kromě samotných definic objektů sdružuje infomace v podstatě o celém projektu. Stěžejním prvkem celého systému je entita *Project*. Jedná se o nejsvrchnější vrstvu v hierarchii jednotlivých objektů, se kterými systém pracuje. Přestože jsou navázané objekty, přesně zařazeny pomocí klíčů, pro zachování přehlednosti a ucelenosti dat je předpoklad, že pro každý projekt ve smyslu jedné zakázky, nebo její části, bude vytvořena nová databáze a tabulka bude obsahovat pouze jeden záznam. Entita *Projet* obsahuje vazby 1:N k entitám *Server*, *ProjectLanguage* a *Vizualization*. Záznamy v tabulce *Server* představují seznam již konkrétních PCL, která jsou v dané zakázce užita. Dané PLC je charakterizováno svým názvem a typem z tabulky číselníku *ServerType*. A jsou mu přiřazeny příslušné technologické objekty z tabulky *Instance* a komunikační rozhraní v tabulce *Bus*. K projektu jsou vázány také užité vizualizace (*Visualization*), která je obdobně jako PLC definována svým názvem a typem (*VisualizationType*). Jazykové varianty projektu jsou dány entitou *ProjectLanguage*, která je tvořena odkazy na příslušný *Projekt* a jazyk z tabulky *Language*. Velmi důležitým objektem databáze je tabulka *MultilingualString*. Jedná se o seznam všech textových položek užitých napříč celým systémem. Velké množství entity obsahuje ve své definici textový popis, jedná se však pouze o odkaz na záznam uložený v tomto seznamu. Důvodem pro tento přístup je využívaný koncept překladů. Každý textový záznam obsahuje svůj překlad, je-li vyžadován, který je definován v tabulce *Translation*. Entity překladu tak obsahují odkaz na příslušný text (*MultilingualString*) a jazykovou variantu (*Language*).

3.1.2 Technologické objekty

Stěžejní částí celé databáze je vytvoření skeletu pro samotnou definici technologických objektů. Centrálním prvkem této části je proto tabulka *Instance*, jejíž záznamy odpovídají objektům dané technologie. K této entitě je navázána celá řada dalších objektů, které utvářejí charakteristiky a vlastnosti daného objektu jako celku. Základní vlastností objektu je přiřazení typu, jehož koncept bude popsán posléze. Dále se jedná o jeho popis, který je sestaven z jednotlivých částí textu nacházejících se ve zmiňovaném seznamu textu. Tento popis je určen záznamy v tabulce *InstanceDescription*, jenž obsahuje odkaz na příslušné entity *MultilingualString*, *Instance* a pořadí v celkovém popisu. Informaci o tom pro, pro které z vizualizací užitých v projektu jsou generový HMI tagy udržuje tabulka *InstanceVisualization*.

Jednou z hlavních vlastností objektu je přiřazení typu, který je představován příslušným záznamem z tabulky *Type*. Jak bylo uvedeno v kapitole 1.2, která se věnuje popisu definiční listiny, typ v podstatě specifikuje obsah daného objektu. Pro specifičtější roztřízení jednotlivých typů, které mohou být definovány, bylo zavedeno třízení typů podle jejich druhů. Jednotlivé duhy typů jsou uvedeny v číselníku s názvem *TypeType*. Zde rozlišujeme následující typy: Instance, Group, GenDrive, ShareDb a ComDb. Díky tomuto třízení, lze rozlišovat definované položky na základě toho, zda odpovídají

konkrétním objektům technologie nebo se jedná o skupinu sdružující tyto objekty. Zda se jedná o objekt, který může obsahovat vnořené položky (vzájemná reference mezi dvěma entitami *Instance*) je dáno právě podle druhu přiřazeného typu. Dále se může jednat o objekty, které odpovídají sdíleným a komunikačním blokům PLC. Typ představuje zapouzdření pro struktury proměnných, které danému objektu náleží a jsou představovány entitami v tabulce *Struct*. Obdobně jako v předchozím případech i zde rozlišujeme, o jaký druh struktury se jedná. Hlavní rozdělení je třízení na struktury typu Setpoint, Command, Status, Fault a dále jsou rozlišovány struktury typu Com, Share a Sub. Zařízení struktury k příslušnému duhu úzce souvisí s typem, kterému náleží. Toto na první pohled složité třízení však značně usnadňuje následnou orientaci v datech. Například typ instančního druhu, může mít přiřazeny až čtyři struktury podle systému, který je blíže popsán v kapitole 1.3 Struktury proměnných instančních datových bloků. Naopak typ objektu, který je zařazen do skupiny pro sdílené objekty, může obsahovat pouze jednu sadu proměnných sdružených ve struktuře, jejíž druh je určen pro proměnné, které jsou učeny pro sdílené bloky. Konkrétní položky proměnných jsou tvořeny záznamy v tabulce *StructItem*, které jsou cizím klíčem přiřazeny příslušné struktuře. Hlavní vlastností proměnných je datový typ. Základní datové typy jsou uloženy v číselníku *BaseDatyType*, Položkou ve struktuře proměnných však může být i odkaz na již vytvořený seznam proměnných, které jsou sdružovány ve struktuře typu Sub (substructure). Ta se pak v podstatě stává datovým typem dané položky a příslušný typ objektu tak obsahuje proměnné i z této substruktury. To vede k eliminování duplicitních definic proměnných, které jsou shodné pro několik typů objektů. Proměnné obsahují popis ve formě odkazu na textový seznam (*MultilingualString*) a mohou mít přiřazeny atributy, které dále specifikují speciální vlastnosti. Atributy jsou tvořeny již zmíněným systémem tabulka entit a tabulka typů (*Attribute* a *AttributeType*). Dále je možné nadefinovat jednotku příslušné proměnné z číselníku *Unit* nebo může být jednotka definována jako závislá na příslušném objektu instance. V takovém případě je informace uchovávaná pomocí tabulky *InstanceUnit*. Na obdobném principu funguje náhrada textových symbolu v popisu proměnných pro konkrétní objekty skrze tabulku *ObjectText*.



Obr. 6 Schéma části databáze pro definici technologických objektů

3.1.3 Generované části

Třetí pomyslný oddíl databáze je závislý již na konkrétním technickém řešení a její část je plněna generovanými daty na základě již vytvořeného řídicího programu PLC programátorem. Pro práci s interlockovými podmínkami jsou definovány tabulky *Interlock* a *InterlockItem*. Entity *InterlockItem* představují jednotlivé interlockové podmínky a jsou přiřazeny příslušné entitě *Interlocku*, která je svázaná s příslušným technologickým objektem. Dále se zde nachází definice komunikačních rozhraní (*Bus, Node, NodeType, Module*) vstupů a výstupů (*Io, IoType*) s vazbami na konkrétní PLC, objekty a textové popisy. Mezi další data, která jsou generována na základě PLC programu patří entity tabulky *LocalVariableText*, jedná se o popisy lokálních proměnných, které jsou definovány programátorem v TIA Portalu.

4. Realizační prostředky pro implementaci aplikace

Náplní této kapitoly je podrobný rozbor jednotlivých prostředků, které byly využity pro vypracovávání zadané práce. Jedná se o vysvětlení pojmů spojených s realizací výsledného systému a uvedení do problematiky jednotlivých oblastí.

Z požadavků uvedených v kapitole 2 věnující se jejich specifikaci, je zřejmé, že cílem bylo vytvořit desktopovou aplikaci pro zařízení využívající Microsoft Windows. Vývojovým prostředím pro vývoj a realizaci výsledné aplikace byl zvolen nástroj Microsoft Visual Studio 2015, který poskytuje rozhraní pro využití objektově orientovaného programovacího jazyka C# založeného na platformě .NET Framework.

Uživatelské rozhraní klientské části je založeno na grafickém subsystému Microsoft Windows Presentation Foundation (WPF), který lze označit jako nástupce staršího konceptu Windows Forms a je součástí .NET Frameworku od verze 3.0 od roku 2007. WPF aplikace je založena na dvou programovacích rozhraních. Prvním je značkovací jazyk XAML, který je vyvíjen společností Microsoft a vychází z jazyka XML. Pomocí tohoto jazyka jsou především definovány a editovány grafické prvky uživatelského rozhraní a jejich funkcionalita je obsluhována pomocí kódu v jazyce C#. Pro práci s databází byl využit již zmiňovaný Entity Framework. V neposlední řadě byl pro naplnění funkcionality systému zahrnut prostředek TIA Openness. [11]

4.1 Model–View–Viewmodel

Hlavním přínosem WPF je možnost oddělení vzhledu a ovládacích prvků od samotného obslužného kódu dané aplikace. Tato vlastnost umožňuje, aby vytvořená aplikace respektovala softwarovou architekturu, označovanou jako Model–View–Viewmodel (MVVM). Tento přístup zvyšuje celkovou přehlednost zdrojového kódu aplikací, poskytuje prostor pro lepší strukturalizaci kódu a umožnuje snadnější implementaci případných změn. Řídicí procesy probíhají na pozadí aplikace, bez přímé vazby na uživatelské rozhraní, které pouze přebírá informace z probíhajících procesů. Průběh jednotlivých procesů je tedy nezávislý na okně aplikace. Tato vlastnost tak umožňuje snadnější sdílení informací mezi více okny. Vzájemné vazby a implementační jazyky jednotlivých vrstev MVVM modelu jsou znázorněny na Obr. 7.

Událostmi řízené grafické rozhraní známé z Windows Forms je zde nahrazeno převážně funkčními prvky *binding* a *command*. Funkcionalita těchto prvků bude popsána níže spolu s podrobnějším rozborem tohoto návrhového vzoru a jeho implementace.

[6][12]

Z názvu architektury je zřejmé, že její struktura obsahuje tři základní vrstvy:

- **Model**

Vrstva představuje předpis pro data, se kterými aplikace pracuje. Vrstva je implementována v jazyce C#. V tomto případě se jedná se o třídy, které prezentují strukturu databázových dat a vycházejí z modelových tříd definovaných v serverové části aplikace. Viz kapitoly 4.3.3 Definice modelových tříd a 5.2.1 Doménové objekty.

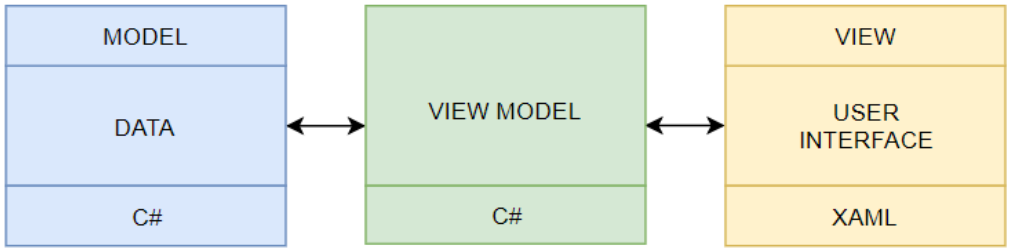
- **View**

Tato vrstva je stěžejní částí klientské části aplikace. Reprezentuje uživatelské rozhraní programu. Jedená se o XAML dokumenty, které definují jednotlivá okna a jejich obsah. Každý xaml dokument je spojen podkladovým C# kódem tzv. CodeBehind, který je zodpovědný za inicializaci objektů. V této části může být alternativně definováno chování jednotlivých prvků. Uživatelské rozhraní se dotazuje příslušné obslužné třídy na aktuální stav a podle něj reaguje a zprostředkovává tak informace uživateli. Tato vazba je obousměrná, provede-li uživatel nějakou interakci je tato událost zpropagována do obslužné třídy.

- **ViewModel**

Jedná se o vrstvu zajišťující tzv. bussiness logiku - obsahuje řídicí procesy aplikace, zpracovává vstupní informace a data, drží stav aplikace a umožňuje propojení vrstvy Modelu a View. Vrstva poskytuje všechna data pro uživatelské rozhraní, které je reprezentováno View vrstvou. Jedná se opět o třídy, které jsou implementovány v jazyce C#. Tyto třídy jsou úzce spojeny s příslušnými ovládacími prvky nebo oknem a mají přístup k třídám představující strukturální model databáze. Z logiky věci vyplývá, že stejně jako View je ViewModel součástí klientské části.

[6][12]



Obr. 7 Znázornění vazeb architektury podle MVVM modelu

4.1.1 Princip datových vazeb

Aby mohlo docházet k přenosu a výměně informací mezi prvky uživatelského rozhraní a zdrojovými daty je nutné zajistit vzájemnou datovou vazbu mezi danými vrstvami. ViewModel je na uživatelské rozhraní navázán tak, že je příslušná instance ViewModelu přiřazena danému View do vlastnosti *DataContext*, čímž je stanoven defaultní datový zdroj vzájemných datových vazeb (*DataBindings*) pro daný XAML dokument. Tyto vazby řídí proces přenosu informací z datového zdroje (obslužná třída ViewModelu) do cíle (View - UI) a naopak. Pro správnou funkci výměny dat mezi vrstvami, je důležité, aby při změně dat ve ViewModelu, byly vyvolány události o jejich změně. Ty umožňují ViewModelu oznamovat změny a uživatelskému rozhraní automaticky zobrazovat aktualizovaná data bezprostředně po jejich změně. Toto chování je podmíněno implementací rozhraní *INotifyPropertyChanged* a *INotifyCollectionChanged*, ta zprostředkovávají událost, která nastane, pokud je některá z vlastností příslušné obslužné třídy změněna. Tato rozhraní jsou implementována v užitých třídách *ObservableObject* a kolekce *ObservableCollection<T>*. Objekt typu *ObserevableObject* je schopen notifikovat změny svých vlastností a zmiňovaná kolekce notifikuje přidání nebo odebrání položek. Díky tomu je uživatelské rozhraní schopno změny zaznamenat a jednoduše tak spravovat a prezentovat data

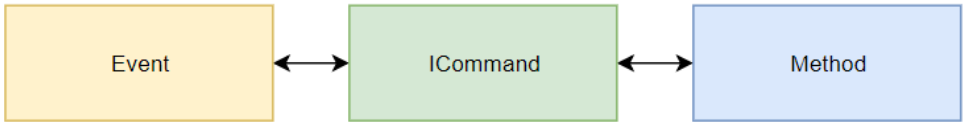
aplikace. Proto lze konstatovat, že tyto třídy v podstatě tvoří základní stavební kameny vrstvy ViewModelu.

Datové vazby jsou zprostředkovávány již zmíněnými elementy *Binding* a *Command*, které poměrně jednoduše specifikují výměnu dat a velice tak pomáhají zefektivnit vývoj aplikaci díky minimalizaci potřebného kódu.

Binding představuje propojení cílového prvku uživatelského rozhraní s konkrétní entitou datového zdroje. Jednotlivé vazby mohou být navázány na specifické vlastnosti cílového prvku. Vlastnosti umožňující toto spojení navázat tak mohu být závislé na vnějším zdroji. Jedná se o vlastnosti typu *DependencyProperty*. Nejjednodušším případem takovéto závislostní vlastností je například vlastnost *Text* u prvku textového pole (TextBox). Datová vazba příslušné vlastnosti úzce souvisí s dalšími prvky, které svým nastavením specifikují její chování a jsou přístupné pomocí klíčových slov.

Vedle *Bindingu* je dalším elementárním prvkem *Command*., neboli příkaz. Ve své podstatě se jedná o metodu ViewModelu, která je skrze rozhraní *ICommand* vyvolávána daným prvkem. Tento prvek (nejčastěji ovládací) má danou metodu přiřazenu ve své vlastnosti nesoucí totožné označení - *Command*. Zda konkrétní prvek disponuje možností přiřazení takové metody je dáno přítomností implementace rozhraní *ICommandSource*. Součástí interface je i vlastnost *CommandParameter*, které umožnuje přiřazené metodě předat parametr. *Command* je zpravidla vázán na událost typickou pro daný prvek (tlačítko - klik). Lze jej však i přiřadit jakékoliv události příslušící vybranému prvku. Schéma navázání události z View do ViewModelu je uvedeno na Obr. 8. Podrobnější popis specifikace vlastností vazeb a užití příkazů je popsán v příloze III.

[6][10][12]



Obr. 8 Schéma navázání události z View do ViewModelu

4.2 TIA Portal Openness

Jak již bylo uvedeno v první kapitole, pro konfiguraci, programování a diagnostiku řídicích systému je využíván software TIA Portal, vyvíjený firmou Siemens, která obsazuje přední příčky na trhu s automatizační technikou. Vývojová platforma poskytuje přehledné prostředí pro správu všech částí projektu.

Součástí Siemens TIA Portal (od verze V13) je nastroj s názvem TIA Portal Openness, který umožňuje propojení vývojového prostředí s externími aplikacemi. Součástí instalace rozhraní TIA Portal Openness, jsou DLL knihovny *Siemens.Engieneering.dll* a *Siemens.Engieneering.HMI.dll*, které představují programátorské rozhraní a poskytují tak přístup k funkcím TIA Portal a umožnují ho vzdáleně ovládat. Díky tomuto rozhraní je možné vytvářet nebo modifikovat projekty, projektová data a v neposlední řadě je exportovat a importovat. Export funkčních bloků a propojovací logiky je realizován ve formě XML souboru.

Po instalaci TIA Openness dochází ve Windows automaticky k vytvoření uživatelské skupiny Siemens TIA Openness, která disponuje právy pro spuštění Openness aplikace a navázání spojení s TIA Portalem. Přidání uživatele do této uživatelské skupiny je danému uživateli umožněno skrze aplikace využívající nástroj Opennes připojení k TIA Portalu

Pro využití funkce importu funkčních bloků a bloků definující typy je nejdříve nezbytné získat soubory, které jsou s touto funkci svázány. Jede o tzv. *Usage-File* a *Enabler-File*, jedná se o XML soubory, které je možné získat skrze technickou podporu firmy Siemens po vyplnění online formuláře. Prvním ze souborů je *SiemensTIAOpennessCustomerID.xml*, jedná se *Enabler-file*, který opravňuje aplikaci k importu souborů do TIA Poratlu. Soubor je nutné umístit do adresáře, kde je umístěn spouštěcí soubor aplikace. Druhým souborem je *SiemensTIAOpennessUsage.xml*, představující *Usager-file*. Soubor umožnuje TIA Portalu využití importovaných bloků a musí být umístěn uvnitř adresářové struktury instalace TIA Portalu, konkrétně pak v …/PublicAPI/V15/... Soubory obsahují informace o majiteli softwaru a jeho identifikaci.

[9][21][22]

4.2.1 Vytvořené obslužné třídy pro TIA Openness

Pro využití produktu Siemens TIA Openness, byla v navrhované aplikaci implementována vlastní vytvořena třída *Openness.cs*, která zapouzdřuje funkce Siemens knihoven pro potřeby komunikace s TIA Portalem včetně jeho obsluhy spolu s exportem a importem dat. Tato třída, včetně tříd nutných pro deserializaci exportovaných dat je součástí vytvořeného projektu TIA, který je komponentou celého řešení.

Zmiňované vývojové prostředí TIA Portal podléhá poměrně dynamickému vývoji, který s sebou přináší každoroční vydání nové verze softwaru. Tyto přechody na využívání stále novějších a aktuálních verzí s sebou však kromě nových funkcí často přináší i mnohá úskalí spojená se zpětnou nekompatibilitou. Tímto faktorem je bohužel zatížen i nástroj Openness. Při vývoji aplikace bylo pracováno postupně s třemi různými verzemi. Během rané fáze testování využití a možností funkcí nástroje byla využívána verze V13, která byla později během vývoje převedena na inovovanou verzi V14. Tato změna přinesla poměrně rozsáhlé změny v zahrnutých *Siemens.Engieneering* knihovách, které vyžadovaly změny syntaxe pro využívání knihovních funkcí. Druhým přechodem byla změna na současnou verzi V15, resp. V15.1. Toto povýšení nevyžadovalo zásadní změny ve struktuře programu, ale přineslo drobné změny ve struktuře generovaných XML souborů, které bylo nutné ošetřit změnami jmenných prostorů pro deserializaci dokumentů.

Jak vyplývá ze zadání, stěžejní funkcionalitou nástroje Openness, která byla v práci využita je export a import funkčních bloků k připojenému k TIA Portálu. Pro tyto procesy byly implementovány funkce, které umožnují využití knihovních funkcí, které jsou k těmto úkonům zahrnuty. Obsah těchto funkcí bude níže popsán. Popisované funkce dále využívají související vytvořené metody například pro procházení a vyhledávání objektů v instanci připojeného TIA portalu.

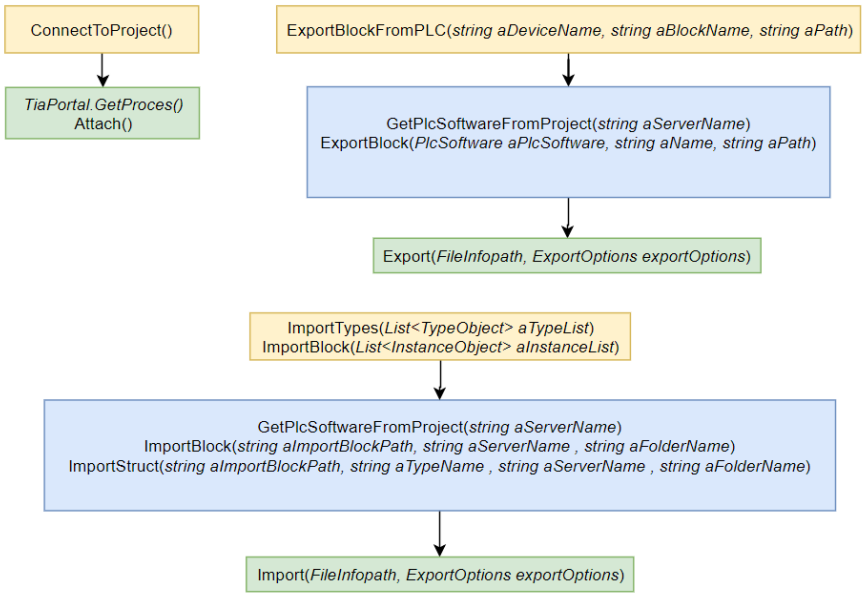
- *ConnectToProject ()* – Jedná se o funkci zprostředkovávající přístup aplikace k danému projektu v TIA Portálu. S využitím funkcí *GetProcesses()* je získán výčet spuštěných procesů TIA Portalu na daném zařízení. Následně volána metoda *Attach()*, kterou je získána příslušná instance TIA Portálu, obsahující kolekci projektů. V této kolekci je zahrnut již konkrétní otevřený projekt a jeho instance je převzata do vytvořené proměnné *OpennedProject*. Knihovní funkce umožnují spustit i zcela novou instanci TIA Portálu

a následně otevřít zvoleny projekt skrze aplikaci, avšak od této varianty bylo vzhledem k praxi upuštěno, neboť na pracovišti je programátorské prostředí spuštěno takřka neustále. Mohlo by tak docházek ke zmatkům například při znovuotevírání totožných projektů v novém okně TIA Poratalu apod. Proto byla ponechána varianta připojení k již spuštěnému TIAPortalu s otevřeném příslušným projektem.

- *ExportBlockFromPLC (string aDeviceName, string aBlockName, string aPath)* – Tato funkce obsluhuje požadavek pro export funkčního bloku zvolené instance. Předávanými parametry jsou název PLC, název instance a cesta pro uložení exportovaného souboru. První fází je tedy nalezení požadovaného PLC. Instance otevřeného projektu *OpenedProject* obsahuje kolekci *Device* a *DevicesGroups*, což je výčet všech hardwarových modulů a složek s těmito moduly v projektu. Jejich postupným procházením je nalezeno požadované zařízení. Dalším krokem je pak nalezení požadovaného bloku v daném zařízení, případně jeho složkách a voláním metody *Export(FileInfo path, ExportOptions exportOptions)* je proveden export. Odtud vyplývá, že každému funkčnímu bloku odpovídá jeden vygenerovaný XML soubor. Výčtový typ *ExportOptions* umožnuje specifikaci exportovaných dat. Pro úspěšný export bloku, je vyžadováno, aby byl příslušný blok tzv. konzistentní, což znamená, že je nutné před exportem bloky zkompilovat.

- *ImportBlock(string aImportBlockPath, string aServerName,string aFolderName)* – Prvním krokem pro import definice funkčního bloku je vytvoření XML souboru s příslušnou strukturou, která je podrobněji popsána v následující kapitole 4.2.2 Struktura XML souboru. Soubor je vytvořen na základě doménového objektu, který vychází z definic vytvořených pomocí uživatelského rozhraní vytvořené aplikace. Stejným způsobem jako v případě exportu je vyhledána příslušná instance požadovaného cílového zařízení, případně složky, do které je metodou *Import(FileInfo path, ImportOptions exportOptions)* importován požadovaný objekt. Výčtový typ *ImportOptions* umožnuje specifikaci importu, v případě existence objektu se stejným názvem.

- *ImportStruct(string aImportBlockPath, string aTypeName, string aServerName, string aFolderName)* – Import struktur proměnných pro funguje obdobným způsobem jako import bloků. Opět je vytvořen XML soubor pro příslušný doménový objekt Typu a jednotlivé struktury proměnných jsou metodou *Import(FileInfo path, ImportOptions exportOptions)* importovány do příslušné složky Typu



Obr. 9 Schéma stěžejních funkcí obalující procesy pro práci s TIA Openness
žlutě – nadřazené metody požadovaného úkonu, modře – pomocné metody,
zeleně-metody knihovny *Siemens.Engieneering*

4.2.2 Struktura XML souboru

Jak již bylo zmíněno, přenos informací mezi aplikací postavenou na programovacím jazyce C# a TIA Portálem je zprostředkováván pomocí XML souborů. Pro konverzi dat ze souboru je využito procesu tzv. deserializace. Tento proces využívá systémových tříd obsažených ve jmenném prostoru *System.Xml.Serialization*. Jedná se o proces, kdy jsou informace z textového XML dokumentu převedeny do příslušných objektových tříd. Struktuře daného souboru tak musí být vytvořeny odpovídající třídy, které představují jednotlivé úrovně resp. elementy XML dokumentu. Tyto elementy jsou nositelem jednak konkrétních dat, ale zahrnují také jejich popis, což znamená hlavně, informaci o jaká data se jedná a případně specifikaci jejich vlastností. Obecně lze říci, že jsou jednotlivé elementy v dokumentu řazeny tak, že vytvářejí hierarchickou strukturu. [14]

V následující části bude podrobněji rozepsána struktura souborů, které jsou vytvořeny s využitím funkcí TIA Portal Openness. Soubor je generován na základě definic jednotlivých funkčních bloků vytvořených ve vývojovém prostředí TIA Portal V15. Tyto soubory jsou poměrně rozsáhlé a běžně zahrnují tisíce až desetitisíce řádků textu značkovacího jazyka. Rozlišujeme dva typy souborů, jednak soubor odpovídající funkčnímu bloku a soubor pro definované struktury. Soubory představující struktury nejsou tak rozsáhlé, nicméně struktura dokumentu je obdobná. Příklad souboru je přiložen v elektronické podobě v příloze.

Soubor odpovídající vždy jednomu funkčnímu bloku obsahuje kořenový element <*Dokument*>, který zapouzdřuje celý XML dokument. Jeho obsahem jsou: <*Engineering*>,<DocumentInfo> a <*SW.Blocks.FB*>. Element *Engieneering* nese informaci o verzi TIA Portalu, *DocumentInfo* sdružuje další vnořené elementy nesoucí informace o dokumentu, kterými jsou například datum vytvoření, typ exportu nebo instalované produkty na daném zařízení.

Stěžejní částí dokumentu je element *SW.Blocks.FB*, jedná se o element charakterizující daný funkční blok. Jeho tělo je tvořeno elementy <*AttributeList*> a <*ObjectList*>. Jak již název napovídá *AttributeList* obsahuje elementy, které nesou informace o vlastnostech bloku. Jedná se například o položky pro název, pořadí, velikost paměti, typ programovacího jazyka a v neposlední řadě element <*Interface*>, kde je zahrnut seznam položek <*Section*>, které odpovídají oddílům proměnných definovaných v TIA Poratalu (proměnné definované typem zvolené struktury, lokální proměnné, konstanty apod.). Uvnitř těchto sekcí jsou pak uvedeny již konkrétní proměnné ve formě elementů <*Member*> se svými parametry udávající název a datový typ. Tyto položky mohou dále obsahovat další vnořené proměnné (viz kapitola 1.3 Struktury proměnných instančních datových bloků).

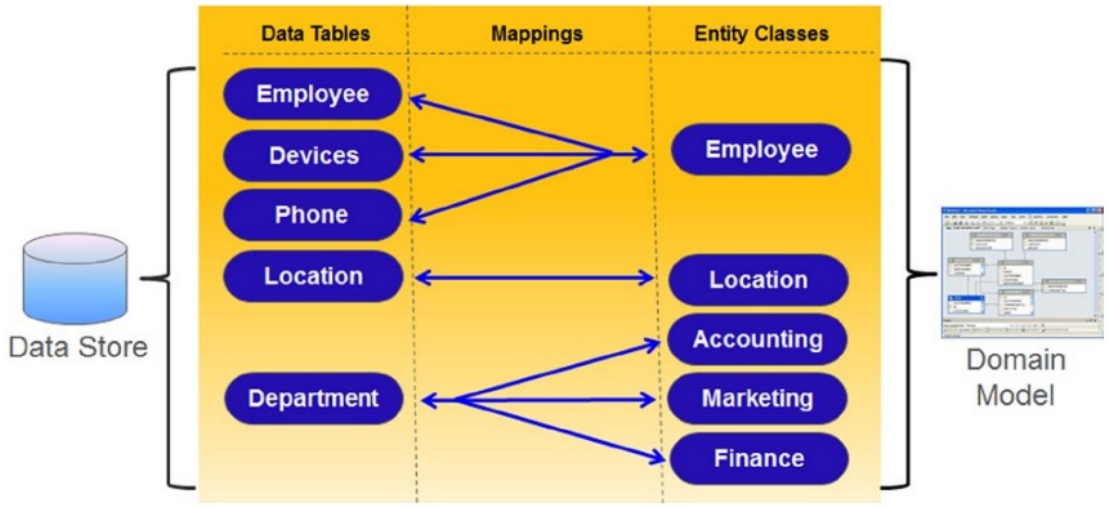
Další část *ObjectList*, obsahuje definici odkazu na textový popis bloku v podobě elementu <*MultilingualText*> a četný seznam položek <*SW.Blocks.CompileUnit*>. Ty představují již konkrétní části programu a logické souvislosti uvnitř bloku. V těle tohoto elementu se obdobně jako v nadřazené úrovni vyskytují položky *AttributeList* a *ObjectList*. *AttributeList* v tomto případě obsahuje <*NetworkSource*>, který obaluje element <*FlgNet*> obsahující části <*Parts*> a <*Wires*>. Tyto elementy definují vnitřní vazby a strukturu FB. Uvnitř části *Parts* jsou zachyceny položky <*Access*>, které představují odkaz na konkrétní proměnnou. Kromě identifikačního čísla je *Access* také nositelem informace, zda se jedná o globální nebo lokální typ proměnné. Její označení je dáno vnořeným elementem <*Symbol*>. Symbol je složen s jednotlivých komponent, ze kterých je sestaven odkaz na tuto proměnnou. Tento odkaz, neboli tag (viz kapitola 1.3), je složen z názvu proměnné a specifikace jejího umístění. Touto specifikací je myšleno případné zařazení proměnné do některé ze skupiny proměnných (Status, Command apod.) a pokud se jedná o globální proměnnou z jiného bloku, tak také názvu příslušného bloku. *Wires* obsahuje definice logických propojení jednotlivých částí a proměnných dané části funkčního bloku. Tato propojení jsou definována skrze numerické identifikátory jednotlivých vazeb. *ObjectList* v této části obsahuje, obdobně jako v předchozím případě, definici pro odkaz popisu daného prvku programu.

U vybraných elementů např. <*Sections*> a <*FlgNet*> je součástí zápisu také jmenný prostor (např. xmlns="http://www.siemens.com/automation/Openness/SW/Interface/v3"0), který udává konkrétní interface pro danou část dokumentu.

[21]

4.3 Entity Framework

Z kapitoly 3 Struktura databáze pro definice objektů, která se zabývá popisem navrhované databáze, je zřejmé, že databáze jako taková je představována tabulkami s jejími sloupci, díky kterým lze definovat vzájemné vztahy pomocí primárních a cizích klíčů. V neposlední řadě je samozřejmě tvořena i obsaženými daty, které představují jednotlivé záznamy. Tyto prvky tak vytváří relační strukturu. Abychom mohli s daty uloženými v databázi pracovat skrze vyvíjenou aplikaci, je nutné zajistit obousměrný přenos dat právě mezi těmito členy systému. Pro převod relační struktury a informací v ní obsažených, do světa objektově orientovaného programovacího prostředí je využito procesu objektového relačního mapování (ORM). Výsledkem tohoto procesu je převod jednotlivých entit databáze na objektové struktury. Tento přístup s sebou také přináší oddělení aplikace od konkrétní databáze což umožnuje snadnější implementaci případných změn na obou stranách. Předpisem pro tyto objekty jsou doménové třídy tvořící doménový model. Doménový model tedy musí obsahovat třídy, které jsou ekvivalentním vyjádřením příslušných tabulek databáze. Vlastnosti těchto tříd pak představují jednotlivé sloupce tabulky. [2][3][17]



Obr. 10 Entity Data Model [3]

Na uvedeném obrázku Obr. 10 Entity Data ModelObr. 10 je znázorněn tzv. Entity Data Model, jedná se o schéma zachycující proces objektového mapování. Data z jednotlivých tabulek nejsou převáděny přímo do tříd, ale jsou mapovány i napříč vzájemnými vazbami mezi tabulkami. Na uvedeném příkladu je zřejmé že objekt třídy *Employee* je sestaven z odkazů na další dvě tabulky *Devices* a *Phone*. Výsledkem mapování je objekt Employee, který však bude obsahovat i příslušné objekty z tabulek *Devices* a *Phone*, případně jejich kolekce jedná-li se o vztah 1:N.

Tento proces ORM je pro aplikace vyvíjené na bázi .NET Frameworku zajišťován právě pomocí open source Entity Frameworku. Tento Framework je nástrojem pro využívání sady tříd ADO.NET, které zprostředkovávají přístup k datovým zdrojům, kterým mohou být právě databáze. Framework jako takový podporuje širokou škálu typů databází jako například Microsoft SQL Server, SQL Server Compact. MySQL Oracle, SQLite, Firebird a mnohé další. Entity Framework je součástí instalace Visual Studia a vyvíjená aplikace využívá současnou verzi Entity Framework 6.2.

[2][3] [18][19]

4.3.1 Přístupy návrhu

Pro obecné použití objektového mapování jsou rozlišovány tři základní principy. Všechny vychází z předpokladu, že celý koncept je založen na třech částech, kterými jsou databáze, model a kód, který s daty pracuje.

- **Database First**

Tento koncept je založen na použití již existující nezávislé databáze. Tabulky jsou mapovány a následně jsou vytvořeny příslušné třídy. Ve Visual Studiu je tímto procesem uživatel provázen pomocí nástroje Entity Data Model Wizard. Třídy mohou být generovány podle šablony a jsou parciální. Což přináší možnost vytvoření vlastních tříd stejného názvu s vlastními metodami a vlastnostmi do stejně pojmenovaných tříd, které v podstatě rozšíří danou třídu. Při opětovném generování modelu pak nedochází ke ztrátám těchto vlastních částí.

- **Code First**

Opačný přístup využívá varianta *Code first*. Prvním krokem při tomto postupu je vytvoření tříd, které definují model a odráží tak strukturu požadované databáze. V projektu je nutné ručně nainstalovat NuGet balíček Entity frameworku, příkazem `install-package EntityFramework` v konzoli pro správu balíčků (Package Manager Console). Při použití *database first* je balíček nainstalován během procesu. Připravené třídy rozlišujeme na entitní třídy, které představují tabulky a jejich sloupce, a třídu pro data kontext. Data kontext třída musí dědit z *DbContext*, která náleží Entity Frameworku. Jednotlivé tabulky požadované databáze jsou v této třídě definovány jako vlastnosti *DbSet<T>*, kde *T* je typ dané entity. Tato třída tak specifikuje, které třídy v konkrétním projektu mají být použity pro vytvoření databáze. Standardně nejsou uváděna pravidla pro generování databázové struktury, proto je pro správné generování primárních a cizích klíčů nutné dodržet konvence pro názvy vlastností. Za primární klíč je zpravidla považována vlastnost s názvem *Id* a cizí klíč je označen *NázevTabulkyId*, případně lze vlastnosti označit atributy. Pravidla pro tvorbu databáze mohou být stanovena v tzv. *code first migracích*. Migrace je proces, kdy jsou nad databází provedeny kroky, jejichž výsledkem je požadovaná struktura databáze. Díky migracím je možné uchovávat jednotlivé verze databázové struktury. Jedná se o mechanizmus, který je schopen vytvořit nebo změnit aktuální verzi struktury již vytvořené databáze na verzi, která odpovídá příslušné verzi představované aktuálním modelem, resp. třídami. Pro implementaci změn tak není nutné vytváření nové databáze. Migrace je vyvolána příkazem v konzoli pro správu balíčků `Add-Migration {NazevMigrace}`. Výsledkem je vygenerování třídy s názvem migrace, která nese informaci o změnách struktury databáze v těle metod *Up()* a *Down()* (přidání nebo odebrání tabulky, sloupců, změn datových typů apod.). To přináší přehled o jednotlivých změnách, které byly během vývoje databáze provedeny, a je možné se k některé z nich opět vrátit. Při vytváření databáze jsou pak postupně provedeny chronologicky všechny změny zachycené ve třídách migrací. V samotné databázi je automaticky vytvořena tabulka *_MigrationHistory*, jejíž záznamy odpovídají jednotlivým migracím, které byly nad danou databází provedeny a seznam tříd ve zdrojové aplikaci musí odpovídat těmto záznamům. Pomocí příkazu `Update-Database`, je možné aktualizovat stav databáze podle poslední, nebo zvolené migrace. S migrací úzce souvisí metoda *Seed()*, která je součástí třídy *DbMigrationsConfiguration<TContext>* . Voláním této metody při vytváření databáze, může být naplněna výchozími hodnotami. Tělo této metody tak může být naplněno podklady pro vzorová data nebo číselníky.

- **Model First**

V této variantě je nejdříve vytvořen model databáze s využitím grafického nástroje pro návrh. Pomocí tohoto nástroje jsou nadefinovány tabulky, jejich vlastnosti, vzájemné vazby a je spuštěno generování jak databáze, tak i příslušných tříd.

[3][17]

4.3.2 Užité návrhové vzory

V této kapitole budou stručně popsány návrhové vzory vztahující se k práci s databázovými daty, které jsou v práci využity. Obecně lze za návrhový vzor považovat principy a postupy, které nabízejí řešení pro danou problematikou a jsou již ověřeny praxí. Za použitý návrhový vzor lze tak považovat i výše zmíněné objektové mapování. Dalšími užitými vzory jsou *Unit Of Work, Repository* a *Identity Map*.

- **Unit of Work**

Návrhový vzor představuje ohraničení jednotlivých kroků, které spolu souvisí a jsou z pohledu uživatele provedeny jako celek. Tento blok operací je označován jako tzv. bussiness transakce. Uvnitř tohoto celku je, v případě práce s databází, s využitím Entity Framework třídy *DbContext* vytvořeno spojení s DB. Během tohoto spojení jsou data získávány, mohou být změna a tyto změny jsou pak promítnuty do úložiště. Při ukládání skrze *SaveChanges()* je pořadí ukládání změn hlídáno, aby nedocházelo ke konfliktním stavům. Poté je spojení ukončeno spolu se zánikem již nepotřebných objektů skrze interface *IDisposable*, aby nedocházelo k zbytečnému zahlcování paměti.

- **Repository**

Jedná se o návrhový vzor pro datovou vrstvu, který představuje třídu, resp. kolekci objektů daného typu, odpovídající zmaterializovaným datům příslušné tabulky. Nad danou kolekcí objektů, je pak možné provádět dotazování a běžné editační operace. Tyto operace jsou označovány jako *CRUD*, což odpovídá zkratce anglických slov *create, read, update* a *delete*. Pro složitější dotazy je možné využít variantu využívající interface *IQueryable*, který navrací příslušný filtrovaný výsledek. Další možnou variantou může být využití *Query objektu*, kdy jsou dotazy nad databází reprezentovány jako třídy, které mohou být parametrizovány a mohou být skrze jednu definici využity opakovaně.

- **Identity Map**

Tento návrhový vzor přísluší situaci, kdy je ke konkrétním datům přistupováno vícekrát. Pokud je v rámci transakce daná entita databáze již materializována, není načtena znovu, ale je použit již existující objekt. Případné změny, které mohou být prováděny ve více krocích, jsou tak podváděny vždy na stejném objektu. V případě absence tohoto přístupu, by byl pro každou následující změnu opět vytvořena nová instance původního, nezměněného, objektu.

[8][13]

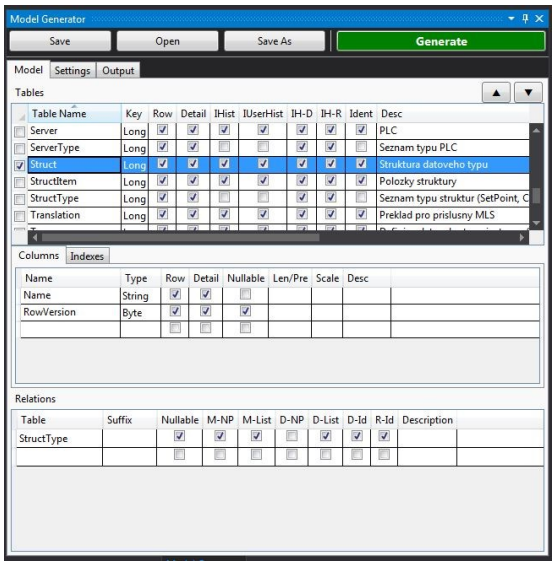
Výše uvedené vzory umožnují jejich použití ve vzájemné kombinaci. Kdy například jedna instance návrhového vzoru *Unit of Work* může uvnitř obsahovat několik instancí vzoru *Repository* a zároveň mohou být uplatněny principy *Identity Map*. Příklad jejich užití je uveden v kapitole 4.3.4, jenž se zabývá dotazováním pomocí technologie LINQ.

4.3.3 Definice modelových tříd

V předcházejících kapitolách pojednávajících o popisu databáze a postupech pro její vznik, bylo zmíněno, že byla vytvořena pomocí přístupu *Code First* vyžadující vytvoření modelu pomocí definovaných tříd. K vytvoření tříd, které databázi předepisují, byl využit nástroj Model Generator. Jedná se o softwarový nástroj pro Visual Studio vyvíjený firmou Ingeteam a.s. Použití tohoto nástroje bylo podmínkou pro dodržení vnitřní architektury vyvíjené aplikace podle firemního standardu.

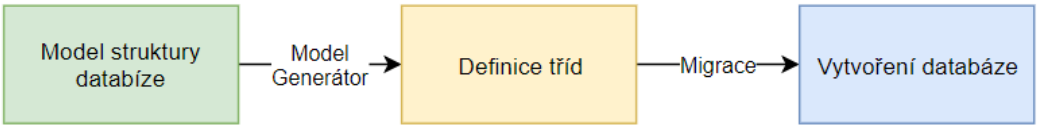
Zmiňovaný nástroj, jehož náhled je zobrazen na níže uvedeném obrázku Obr. 11, umožňuje velmi efektivní a přehledné vytvoření pomyslného modelu jednotlivých tabulek databáze, relačních vztahů mezi nimi a jejich sloupců. V prvním okně *Tables* jsou definovány jednotlivé tabulky, je stanoven typ primárního klíče a zda se jedná o unikátní klíč. Jedná-li se o unikátní klíč (políčko *Ident*) je databází pro každý nový řádek tabulky automaticky inkrementován. V opačném případě je možné jej zadávat ručně, například při plnění některých částí databáze, kterými mohou být výčtové seznamy. Dále je možné

nastavit, jaké další části mají příslušné automaticky generované třídy obsahovat. Jedním z příkladů může být Interface pro obsluhu záznamů historie změn jednotlivých záznamů. Následuje okno *Columns* pro definování sloupců a jejich vlastností.



Obr. 11 Náhled okna nástroje Model Generator

V poslední části *Relations* jsou vystavěny vazby mezi jednotlivými tabulkami a možnost volby pro generování navigačních vlastností mezi danými třídami a kolekcí pro referencemi svázané objekty. Nástroj umožnuje uložení modelu definic tříd pro další užití ve formě XML souboru. Proces vytvoření databáze s využitím Model Generátoru je vyobrazen na blokovém schématu, které je uvedeno na Obr. 12.



Obr. 12 Blokové schéma postupu vytvoření databáze

Na základě takto nadefinovaného třídního modelu databáze, je možné vygenerovat příslušné třídy s odpovídajícím obsahem pro příslušné části celého řešení ve Visual Studiu. Jednotlivé části řešení, jejich obsah a funkcionality jsou podrobněji rozebrány v kapitole 5.1, která se věnuje právě struktuře aplikace. Kromě tříd spojených s modelem databáze, jsou tímto nástrojem generovány i další třídy označované jako *Controllery* a kostry pro příslušná rozhraní, která implementují. Význam těchto tříd je objasněn ve výše zmiňované kapitole.

Každé tabulce přísluší tří vygenerované typy tříd. Nejjednodušším typem je základní třída označována příponou v názvu *Row*, jedná se třídu kopírující sloupce tabulek. Dalším typem je třída s příponou *Detail*, oproti předcházejícímu typu je doplněna o kolekce objektů, které obsahují v cizím klíči referenci na tento objekt. Příkladem muže být objekt struktury (*StructureDetail*), obsahující si

kolekci příslušných položek (*StructItemDetail*). Nejobsáhlejší třídní strukturou jsou bázové třídy, které přesně odráží všechny vlastnosti objektu vyplývající z definice tabulek vazeb mezi nimi. Třída obsahuje oboustranné navigační vlastnosti, které obsahují objekty, jenž jsou vzájemně svázány cizími klíči. Jedná se o objekty, které obsahují referenci na daný objekt, ale také objekty, na které je daný objekt referencován. V případě výše zmiňovaného příkladu tak objekt *StructItem* bude obsahovat kromě *Id* nadřazené struktury, také celý konkrétní objekt této struktury. Objekty jsou tak schopny reflektovat výsledek objektového mapování databáze.

4.3.4 Skládání LINQ dotazů

Kromě vytvoření samotné databáze a navázání spojení, je nutné také zajistit možnost s danými daty manipulovat. Tím je rozuměn nejen o přístup ke specifikovaným datům za účelem jejich získání, ale také jejich modifikace, vkládání nebo mazání. Této možnosti je dosaženo pomocí technologie Language-Integrated Query, označované jako LINQ, která je zprostředkována .NET Frameworkem.

Dotazování na zdroj dat je zprostředkováváno, pomocí implementace rozhraní *IQueryable<T>*, které je generickým typem a je součástí jmenného prostoru *System.Linq* Frameworku .NET. Toto rozhraní je rozšířením rozhraní *IEnumerable<T>*, díky kterému je získaná data možné reprezentovat ne jako jeden objekt, ale jako jejich kolekci. Princip dotazování je takový, že pomocí LINQ jsou nadefinovány podmínky konkrétního dotazu, které jsou pak přeloženy na řetězec, představující předpis pro dotaz již v jazyce SQL. Dotaz je vykonán na příslušné SQL databází a jsou navrácena data splňující dané podmínky. Nadefinovaný LINQ dotaz pracuje na principu tzv. *lazy loadingu*, kdy je dotaz nad databází vykonán až v momentě, kdy je požadovaný objekt nebo prvek potřeba. LINQ můžeme využít nejen při vytváření dotazu nad databázovými daty, ale v podstatě nad jakoukoliv kolekcí implementující rozhraní *IEnumerable<T>*. To umožnuje využití funkcí třídy *Enumerable* pro výběr, třízení a filtraci dat. Příkladem mohou být funkce *Where*, *OrderBy* nebo *First*, kdy jsou do argumentu těch to funkcí vloženy podmínky pro výběr. Na níže uvedeném příkladu bude popsán konkrétní případ užití této technologie pro dotazování nad databází. [16]

Příklad uvedený na Obr. 13 zachycuje jednoduchou metodu datové vrstvy aplikace, která je součástí třídy zajištující práci s daty nad entitou databáze *Interlock*, konkrétně smazání záznamu v tabulce. V metodě je využit návrhový vzor Unit of Work, představovaný blokem *using*, který deklaruje lokální proměnnou *lUnit* představující spojení mezi aplikací a databází. Spojení je definováno pomocí třídy *DataContext*. Ta je obsažená ve frameworku nesoucí označení *FX*, který je vyvíjeném firmou Ingeteam a.s. Ten některé z funkcionalit EntityFrameworku obaluje a rozšiřuje. V tomto případě se jedná o rozšíření třídy *DbContext*. Dále následuje definování LINQ dotazu pro danou databázi, pomocí vlastnosti *Query*, která implementuje výše zmiňované rozhraní *IQueryable<T>*. *T* představuje entitu, resp. kolekci, nad kterou je dotaz volán, v tomto případě se tedy jedná o objekty třídy *Interlock*. Skrze metodu *Where* je definována podmínka pro dotaz. Syntaxe podmínky využívá lambda výraz. Podmínka je v tomto případě stanovena na navrácení objektu, jehož *Id* je shodné s vlastností *Id* náležící objektu, který je předáván v parametru popisované metody. Návratovým typem funkce *Where* je *IQueryable<T>*. LINQ však umožnuje řetězení jednotlivých metod, proto může být za definicí dotazu připojena další metoda materializující výsledek dotazu. V tomto případě se jedná o metodu *First()*, jenž navrací první položku kolekce objektů splňujících danou podmínku. Protože se v tomto dotazu podmínka týká primárního klíče tabulky, je z logiky věci jasné, že výsledná kolekce bude obsahovat

pouze jeden objekt. V případě očekávaného výsledku obsahující více položek, lze využít metodu *ToList()*. Na závěr bloku je vybraný objekt, odstraněn z databáze skrze metodu *Delete()* a změny jsou uloženy voláním bezparametrické metody *Save()*. Blok operací je v tomto případě definován pomocí třídy stejného názvu *UnitOfWork*, která se stará za navázání spojení implementuje, již dříve zmiňované rozhraní *IDisposable*, a proto je po ukončení všech operací umožněn zánik všech nepotřebných objektů.

```
public bool DeleteInterlock(ICallContext aContext, InterlockDetail aInterlock)
{
    try
    {
        using (var lUnit = UnitOfWork.Get<DataContext>())
        {
            if (aInterlock.Id > 0)
            {
                var lQuery = lUnit.Query<Interlock>().Where(a => a.Id == aInterlock.Id).First();
                lUnit.Delete(lQuery);
                lUnit.Save();
            }
        }
        return true;
    }
    catch (Exception lEx)
    {
        mLog.Error(lEx);
        return false;
    }
}
```

Obr. 13 Příklad práce s databázovými daty pomocí LINQ dotazu s využitím Unit of Work

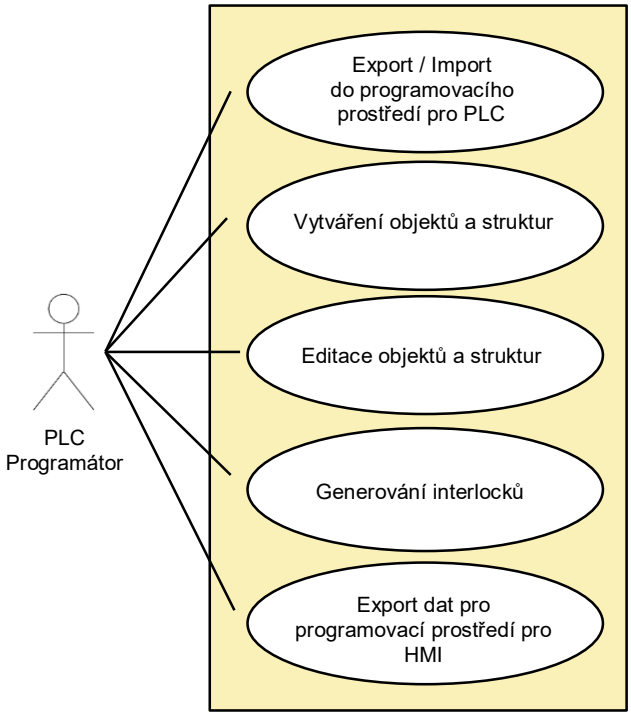
Uvedený blok kódu obsahující LINQ dotaz je ekvivalentem k příkazu v jazyce SQL:

```
DELETE FROM Interlock
WHERE Id == aInterlock.Id
```

5. Realizace aplikace

Následující podkapitoly budou věnovány bližšímu popisu výsledné aplikace, její vnitřní architektuře spolu s mechanismy využívaných pro zpracování vstupních dat a uživatelskému rozhraní.

Hlavním výstupem této diplomové práce je realizace softwarového nástroje, který představuje mezivrstvu pro vývoj řídicího programu pro technologie řízené pomocí PLC. Po rámcovém návrhu databáze tak byla dalším krokem implementace samotné aplikace umožňující práci s databázovými daty a splnit požadavky, kterým je věnována kapitola 2.1 Funkční požadavky. Vytvořená aplikace tak představuje pro uživatele rozhraní, umožňující definici objektů řídicího systému a jeho struktur na základě reálných vlastností řízené technologie a ze stanovených požadavků na daný systém. Je tedy nástrojem pro vytvoření a správu databáze, ve které jsou ukládány definice těchto prvků. Aplikaci tak tvoří část pro komunikaci a práci s daty v SQL databázi s část umožňující tato data interpretovat ve vhodné podobě uživateli. V předcházejících kapitolách bylo již uvedeno, že realizovaná aplikace poskytuje také možnost komunikaci s TIA portálem. Tato komunikace umožňuje obousměrnou výměnu dat mezi databází a programátorským prostředím s využitím platformy TIA Openness. Mimo jiné lze pomocí aplikace generovat zdrojová data pro vizualizace řídicích procesů. Tato data jsou generována na základě exportovaných dat právě z TIA Portalu. Funkcionalita vytvořené aplikace je shrnuta na diagramu užití, který je znázorněn na Obr. 14.



Obr. 14 Diagram užití vytvořené aplikace

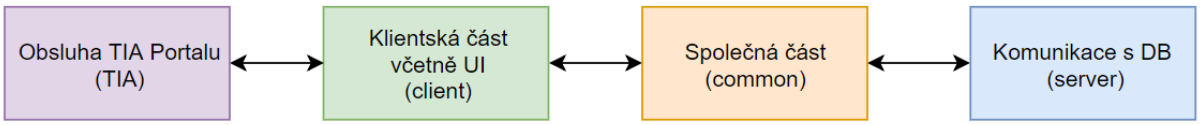
5.1 Vnitřní struktura aplikace

Protože vytvářený systém je poměrně rozsáhlým a komplexním celkem bylo nutné již při návrhu a jeho následné programové implementaci vhodně vytvářet dílčí celky. Toto rozdělení na jednotlivé

komponenty a vrstvy systému s sebou přináší značnou přehlednost vnitřní struktury aplikace. Celé řešení je složeno ze čtyř projektů.

Základní části tvořící architekturu výsledné aplikace vychází ze standardu používaného ve firmě Ingeteam a.s. Jedná se o tři vrstvy: *Client, Common* a.*Server*. Vazby a vzájemná funkcionalita těchto elementů jsou hluboce provázány s již dříve zmiňovaným *Ingeteam.FX* frameworkem, který je majetkem firmy Ingeteam a.s. Všechny tyto vrstvy obsahují příslušné třídy odrážející strukturu databáze. Tato struktura také úzce souvisí s použitím nástroje Model Generátor, který dané třídy na základě definice vytvoří (viz. kapitola 4.3.3 Definice modelových tříd). Pro úkony spojené s využitím nástroje TIA Openness je součástí řešení vytvořený projekt s názvem TIA. Vzájemné vazby mezi jednotlivými komponenty, které spolu funkčně souvisí, jsou znázorněny na schématu uvedeném na Obr. 15 Komponenty tvořící strukturu aplikace.

V následujících částech této podkapitoly bude uveden podrobnější popis vysvětlující funkce a obsah jednotlivých celků.



Obr. 15 Komponenty tvořící strukturu aplikace

- **Client**

Komponenta označována jako klientská část je z pohledu uživatele v podstatě základním prvkem celého systému. Zpracovává surová data, která jsou jí předávány z okolních části. Pro práci s daty byla implementována architektura podle návrhového vzoru fasáda, která zjednodušuje prezentaci dat. Podrobnému rozboru a popisu vnitřní architektury této části je věnována níže uvedená kapitola 5.2 Architektura pro práci s daty. Přestože je tato část díky vzájemným referencím schopna přímo přistupovat na modelové třídy ve společné části, byla vytvořena vrstva doménových objektů. Jedná se o objekty odrážející vlastnosti modelových tříd spolu s doplňujícími vlastnostmi a metodami. Popisu doménových objektů je věnována samostatná podkapitola 5.2.1 Doménové objekty. Tím je zajištěno ještě kompletnější oddělení mezi objekty generovaných tříd, které souvisejí přímo s modelem databáze a mezi objekty jejíž data jsou zpracovávány a modifikovány na úrovni uživatelského rozhraní.

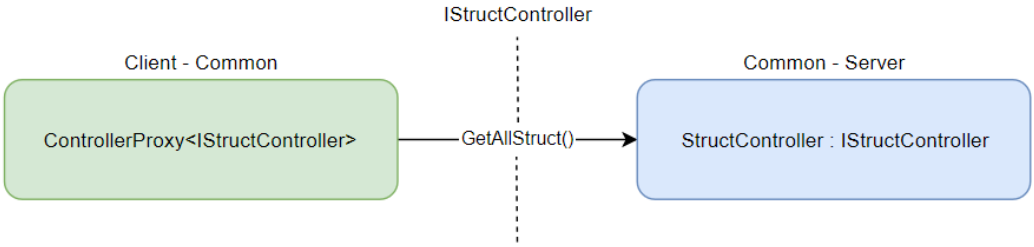
Právě tento element je poskytovatelem uživatelského rozhraní. Pro jeho konstrukci a obsluhu se zde nacházejí příslušné soubory, na základě principu popsaného v kapitole 4.1 Model–View–Viewmodel. Jsou jimi například XAML soubory pro vizualizaci rozhraní, odpovídající třídní modely, konvertory a další podklady.

- **Common**

Jak již název a výše uvedené schéma na Obr. 15 napovídá, jedná se o spojující prvek mezi klientskou částí a částí určené pro obsluhu databáze. V této části se tak nachází třídy, které jsou přístupné pro oba zmiňované celky. Mimo pomocné třídy se jedná hlavně o modelové třídy typu *Row* a *Detail*. Mezi pomocné třídy řadíme vytvořené obslužné rutiny událostí, definované výčtové typy pro číselníky a objekty pro přenos dat (DTO). Dále je zde implementován společný interface pro vzájemnou výměnu

dat. Tento interface je označován jako *IController* a je definován pro každý databázový objekt pomocí Model Generátoru. Interface je aplikován na straně serverové části příslušnou třídou *Controlleru*. Na straně klienta je interface využíván skrze generickou třídu *ControllerProxy<T>*, jenž je součástí frameworku *Ingeteam.FX*. Každému objektu databázové struktury tak náleží metoda pro získávání dat, interface a příslušný objekt pro jejich obsluhu. Toto rozdělení pro jednotlivé objekty přináší do přehlednost a snadnou orientaci v programu, protože lze sdružovat logicky související dotazy nad databází na konkrétním místě. Zjednodušuje se tak i implementace případných změn. Vzhledem k rozsáhlosti databáze je existence pouze jedné sady těchto komponent takřka nepředstavitelná.

Princip funkce výměny dat je takový, že klient je schopen pomocí *ControllerProxy* zprostředkovat volaní metod příslušného *Controlleru* na serverové části. Daná metoda musí být samozřejmě implementována v příslušném *IControlleru*. Uvnitř metod *Controlleru* jsou vykonány příslušné operace pro získání požadovaných dat nebo objektů z databáze. Ty jsou posléze navráceny zpět opět skrze *ControllerProxy*. Díky tomuto systému jsou eliminovány jakékoliv pevné vazby mezi serverovou a klientkou částí. Vždy se jedná o vazby *Client-Common* a *Common-Server* jak je uvedeno na Obr. 16.



Obr. 16 Schématické znázornění principu funkce *controlleru* pro objekt struktury

- **Server**

V předcházejícím textu již byla nastíněna funkce této komponenty, které spočívá již v přímé komunikaci s databázovým serverem s využitím Entity Frameworku. Plní tak úlohu poskytovatele surových dat pro další zpracování ve vyšších vrstvách aplikace. Jedná se o oddělenou část, která běží autonomně jako proces na pozadí, se kterou klient komunikuje. V současné podobě se jedná o komunikaci v rámci jednoho zařízení (localhost). Toto umístění umožnuje připojení jak na vzdálené databáze, tak i na lokální SQL server v daném zařízení. Jednou z možných variant je také umístění společné serverové části pro všechny uživatele do místa úložiště vzdálené databáze. V takovém případě by uživatel ztratil možnost připojení do lokální databáze, což by opět vyžadovalo spuštění vlastní instance serverové aplikace konkrétním zařízení. Samotná problematika týkající se práce nad databázovými daty z více míst, jejich ukládání do lokálních databází a následná synchronizace je rozsáhlý okruh problémů, které budou předmětem dalšího rozvoje a zdokonalování navrhovaného systému.

Nastavení připojení na konkrétní databázový server je dána konfiguračním XML souborem. Tento soubor je zdrojem pro třídu *ConnectionConfig.cs*, která je součástí *Ingeteam.FX* frameworku. Tato třída je využita obslužnými třídami zajišťující připojení s databázovým serverem, které dědí ze třídy *DbContext* náležící *Entity Frameworku*.

Základními parametry pro připojení jsou:

Provider – druh poskytovatele databázové platformy.
Host – umístění SQL serveru.
Catalog – název konkrétní databáze.
User – přihlašovací údaje.
Password – heslo.

Protože je tato část zodpovědná za obsluhu databáze nachází se zde třída *DataMigrationConfiguration.cs*, která implementuje metodu *Seed()*, jenž obsahuje definice výchozích dat pro jednotlivé číselníky, kterými je databáze naplněna již při vytvoření. Stěžejní část tohoto celku jsou *Controller* třídy. Ve výše uvedeném textu popisující společnou část struktury aplikace bylo zmíněno, že se jedná o třídy obsahující metody pro komunikaci s databází. Jejich účelem je získávání požadovaných dat a jejich vkládání. Nachází se zde tedy už konkrétní LINQ dotazy. Podrobnější popis týkající se práce s databází a přenosu dat je uveden v kapitole 4.3 Entity Framework.

- **TIA**

Tato část je vyčleněna pro práci s TIA portálem a obsahuje příslušné vytvořené třídy, které jsou popsány v kapitole 4.2.1 Vytvořené obslužné třídy pro TIA Openness. Dále se zde nachází pomocné třídy pro deserializaci XML dokumentů exportovaných z TIA Portálu a třídy pro transformaci exportovaných dat. Ty zpracovávají data do vhodné podoby pro využití v obslužných vrstvách klientské části aplikace.

5.2 Architektura pro práci s daty

Surová data z databáze jsou systémem přijata ve formě příslušných tříd. Pro jejich zpracování a samotnou distribuci uživateli byla v klientské části aplikace vytvořena architektura, která celý tento proces zaštiťuje. Jejímu popisu je věnována právě tato podkapitola.

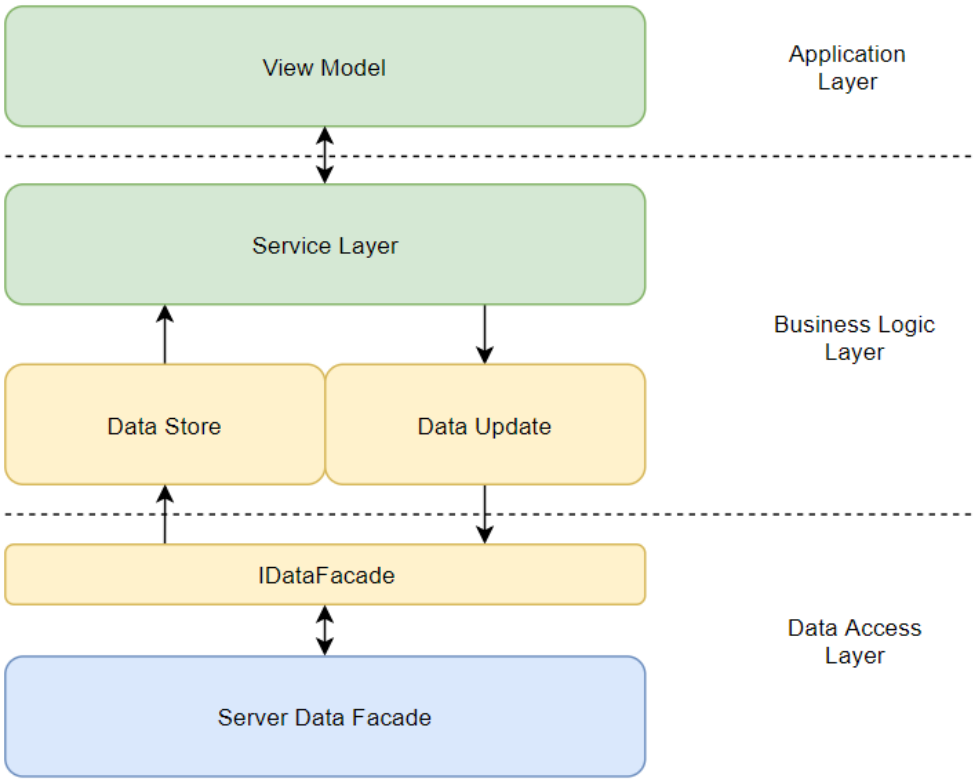
Základní architektura spočívá v rozdělní systému zpracování dat do jednotlivých vrstev, které pracují s daty na odlišných úrovních. Pro implementaci systému a jeho rozdělení na dílčí subsystémy byla využita struktura vycházející z návrhového vzoru označovaný jako fasáda. Fasáda definuje rozhraní, které poskytuje jednodušší a obecnější přístup k jednotlivým prostředkům a vrstvám systému. Požadavek z nadřazených subsystému je tak skrze objekt datové fasády delegován na odpovídající třídu nebo metodu v nižší vrstvě systému. [4]

Proces zpracování dat je rozdělen do tří vrstev, které jsou zobrazeny ve schématu na níže uvedeném Obr. 17. Nejnižší úroveň, označena jako *Data Access Layer*, představuje vrstvu pro práci se surovými daty. Jedná se o vstupní a výstupní bod klientské části pro výměnu dat s databází. Prostřední vrstvu tvoří tzv. *Business Logic Layer*. Tato část systému obsahuje obslužné metody pro práci s daty. Zde jsou data přijatá datovou vrstvou modifikována a ve vhodné podobě interpretována nejsvrchnější vrstvě s názvem *Application Layer*. Ta je určená pro prezentaci a úpravu dat skrze uživatelské rozhraní. Obdobně systém funguje i v opačném směru, například při procesu ukládání. V takovém případě jsou data z nejsvrchnější vrstvy postupně předány datové vrstvě. Během tohoto procesu jsou samozřejmě postupně transformovány do vhodné podoby, aby mohly být uloženy do databáze.

Jednotlivé vrstvy jsou prezentovány příslušnými skupinami tříd. Nejvyšší kategorii tvoří již dříve zmiňované třídy ViewModelu, ty pracují s finální podobou dat prezentovaných uživateli. Nachází se zde hlavně komponenty spravující uživatelské rozhraní jako takové. Většina operací týkajících se práce

s daty je delegována na obslužné metody obsažené v třídách skupiny *ServiceLayer*. Mezi tyto operace patří například požadavek stažení dat z databáze, uložení dat, vytváření objektů, modifikace objektů vyžadující složitější logické operace, operace nutné k promítnutí změn apod. Jedná se o statické třídy, díky kterým jsou vytvořeny soubory metod pro práci s daty dostupné z různých míst systému. Třídy jsou rozděleny podle logických souvislostí. *InstanceServiceLayer.cs* pro funkce spojené s definicemi objektů, *InterlockServiceLayer.cs* pro úkony spojené s interlocky, *TypeServiceLayer.cs* pro úkony spojené s definicemi struktur. Dále pak *ServiceLayer.cs* pro ostatní úkony jako například nastavení konfigurace a *TiaServiceLayer.cs* zprostředkovávající všem nadřazeným částem systému přístup k instanci připojeného TIAPortlu a souvisejícím metodám.

Některé funkce jednotlivých *ServiceLayer* tříd pracují lokálně, což znamená, že je vstupní požadavek zpracován na výstup v rámci dané třídy. Příkladem mohou být filtrace číselníků. Většina metod však pro svou funkci vyžaduje přístup k dalším datům. Poskytovatelem a ve své podstatě i uložištěm je skupina tříd *DataStore*. Jedná se o místo uložení lokálních dat systému. Na základě požadavku jsou vyšším obslužným metodám navrácena odpovídající lokální data nebo aktuální data z databáze. Systém lokálních dat a jejich verzování je podrobně rozebráno v kapitole 5.2.2 Systém lokálních dat a jejich verzování. Pro požadavky ukládání dat slouží třídy skupiny *DataUpdate*. Zmiňované skupiny jsou součástí spojení mezi obslužnou a datovou vrstvou systému. Každá obsahuje tři třídy a jsou opět rozděleny podle toho, zda se jedná o data struktur, objektů nebo obecná.



Obr. 17 Schéma datových vrstev a skupin příslušných tříd

Spojení datové a obslužné vrstvy je zprostředkováno pomocí rozhraní *IDataFacade*. To je implementováno skupinou tříd datové vrstvy s názvem *ServerDataFacade*. Obě komponenty jsou jako v předchozích případech vytvořeny ve třech příslušných variantách. Datové třídy pak skrze *ControllerProxy* komunikují se serverovou částí aplikace a databází (viz. 5.1 Vnitřní struktura aplikace).

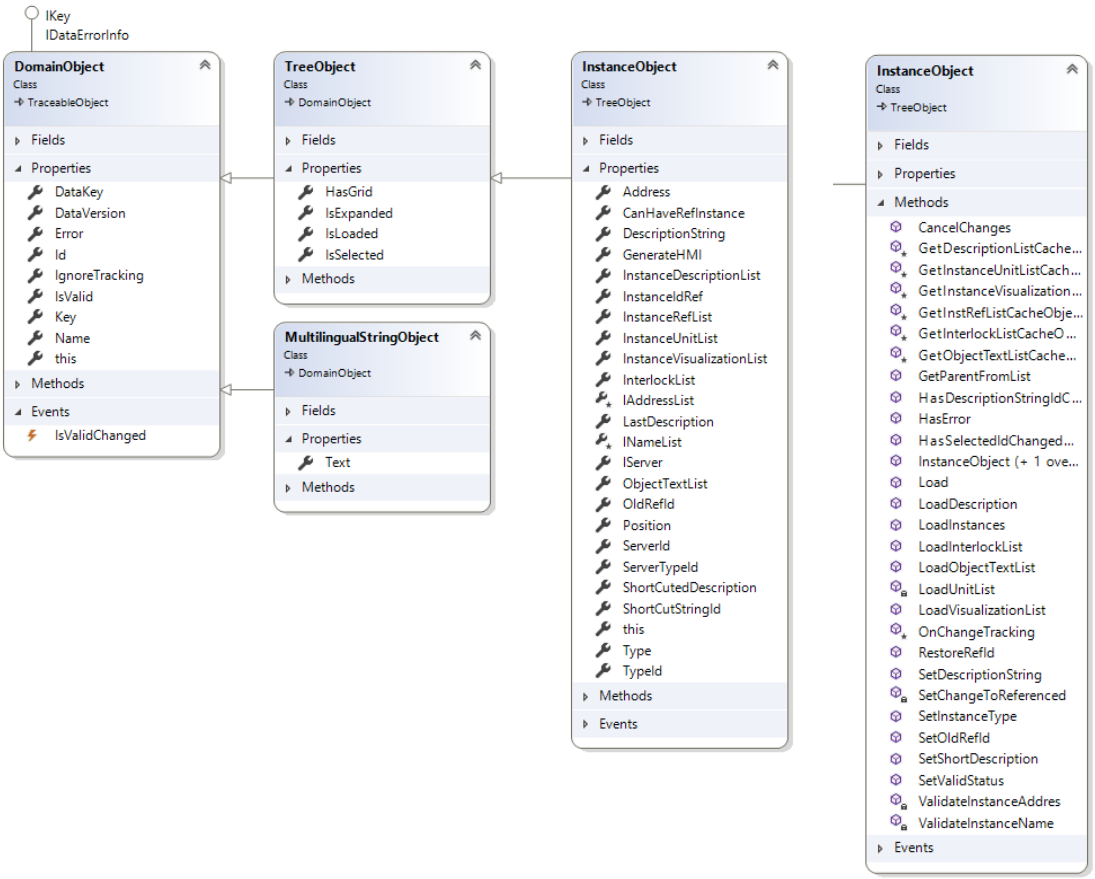
Toto rozdělení na malé a oddělené celky umožnuje implementaci změn na jednotlivých částech bez nutnosti změn na zbytku systému.

5.2.1 Doménové objekty

Pro prací s daty a jejich prezentaci uživatelskému rozhraní byla vytvořena sada objektů vycházející vlastnosti modelových tříd. Každé modelové třídě zastupující databázový objekt tak byla definována také třída ze skupiny doménových objektů. Např. *StructItemObjec.cs*. Tyto třídy jsou rozšířeny o vlastnosti a metody, které napomáhají interpretovat databázová data do finálních podob definic objektů a struktur se kterými pracuje uživatel. Jsou definovány a využívány pouze v klientské části aplikace.

Pro zjednodušení a zobecnění definic doménových objektů bylo využito principu dědičnosti a polymorfizmu. Výchozí třídou je *DomainObject.cs* zaštituje vlastnosti společné všem objektům, jako například název, Id nebo vlastnosti a metody pro sledování změn a kontroly validace dat. Potomky této třídy jsou už konkrétní objekty pro příslušné modelové třídy. Speciální skupinu tvoří třídy, jejíž objekty jsou využívány pro hierarchické zobrazování v uživatelském rozhraní. Jejich rodičem je třída *TreeObject.cs* , které také vycházející ze základního doménového objektu. Třída definuje chování při řazení položek do stromové struktury. Její přidanou hodnotou je možnost tzv. *Lazy Loadingu*, jedná se o proces, kdy jsou data z databáze vyčítána až na základě požadavku vzniklého potřebou dat. Princip je takový, že ve výchozím stavu jsou načteny kořenové položky stromové struktury, avšak jejich detailnější obsah může být doplněn až v případě potřeby těchto dat, například při rozbalení položky. K tomu to účelu slouží virtuální metoda *Load()*, jejíž tělo je specifické pro každý typ potomka. Na uvedeném Obr. 18 je znázorněn třídní diagram pro dva typy doménových objektů. Prvním z nich je *MultilingualStringObject.cs*, jedná se takřka o nejjednodušší objekt mající pouze jednu specifickou vlastnost *text*. Druhým příkladem je třída *InstanceObject.cs*, jedná se naopak o jeden z nejrozsáhlejších typů objektu s mnoha vlastnostmi a metodami pro specifikaci technologického objektu.

Ke konverzi dat z databázových objektů na doménové a opačně, dochází ve skupinách tříd *DataStore* a *DataUpdate*. Tento proces je zajištěn pomocí objektu statické třídy konvertoru, který je součástí *FX* frameworku. Ten pomocí reflexe kopíruje obsah shodných vlastností.

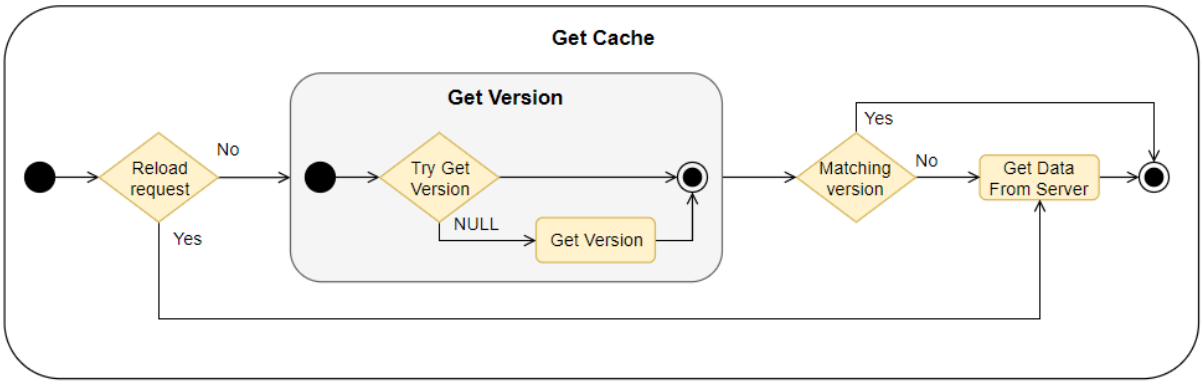


Obr. 18 Diagram tříd zachycující strukturu dědičnosti doménových objektů

5.2.2 Systém lokálních dat a jejich verzování

Pro optimalizaci počtu opakovaně navazovaných spojení mezi databázovým serverem a samotnou aplikací byl implementován systém pro ukládání již vyčtených dat do lokální mezipaměti, tzv. cache. Jedná se o systém umožňující rychlejší a efektivnější přístup k datum. Požadovaná data jsou přijata z databáze a uložena skrze vytvořenou třídu *ObjectCache.cs*, obsahující metody *GetCache()*, *Update()* a *Clear()*. Mezi vlastnosti patří název, podle kterého jsou data identifikovatelná, verze dat, funkce pro získání dat a samotná data. Datový typ ukládaných dat je definován pomocí genericity. Identifikátor cache je ve většině případů, s výjimkou číselníků, složen z názvu odpovídajících dat, a Id příslušného objektu – např. *StructItemListStructId10*, což odpovídá listu položek proměnných pro strukturu s Id 10. Pro správnou funkci tohoto mechanismu je nezbytný modul pro správu verze dat. Díky verzi v podobě číselného údaje lze identifikovat, zda jsou data v mezipaměti aktuální ve srovnání s daty uloženými v databázi. Během vytvoření cache je do kolekce slovníku přidána položka identifikátoru s přiřazenou počáteční verzí č.1. Při ukládání dat příslušné cache do databáze je současně navýšena i její verze. V případě požadavku na konkrétní data v mezipaměti je porovnána lokální verze s poslední uloženou a v případě neshody jsou data aktualizovány. O správu verzí obsluhuje pomocná třída *VersingModule.cs*, jenž obsahuje zmiňovaný slovník verzí a metody *GetVersion(string aCacheName)*, *CreateVersion(string aCacheName)*, *Increment(string aCacheName)* a *RefreshRequestForAll()*, která

nastaví všechny cache na počáteční verzi a v případě požadavku dat vyvolá jejich aktualizaci. Aktualizaci pouze pro konkrétní sadu dat, lze vyvolat předáním parametru pro žádost obnovení. Princip systému získávání dat je zachycen na diagramu aktivit znázorněném na Obr. 19.



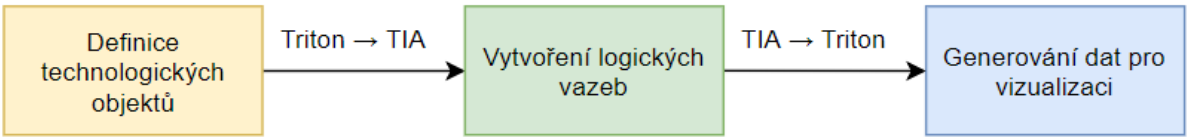
Obr. 19 Diagram aktivit pro získávání dat z paměti

Díky tomuto mechanismu lze opakovaně použít již stažená data bez potřeby navázání spojení s databázovým serverem což optimalizuje časovou náročnost operací. V případě jejich lokálních změn jsou modifikovaná data přístupné i dalším částem obslužné vrstvy aplikace ještě před samotným uložením do databáze.

5.3 Popis uživatelského rozhraní a kroky užití

V předcházejících kapitolách již bylo zmíněno, že uživatelské rozhraní pro obsluhu navrhovaného systému pro definice technologických objektů je založeno na platformě WPF. Ta využívá strukturu oddělených vrstev pro definice zobrazovacích a kontrolních prvků a vrstev pro obsluhu jejich funkcionality. Principu, jakým je UI vystavěno, je podrobněji věnována kapitola 4.1 Model–View–Viewmodel. V následujícím textu budou popsány stěžejní prvky vytvořeného uživatelského rozhraní spolu s popisem jednotlivých úkonu při užívání aplikace.

Využití navrhovaného systému je rozděleno do několika kroků. Prvním z nich je samotná definice technologických objektů. Následně je vytvořen řídicí program PLC programátorem v TIA Portal. Na základě vytvořených logických vazeb mezi objekty a jejich proměnnými jsou pak pomocí aplikace Triton generovány podklady pro vizualizaci.



Obr. 20 Schéma fází procesu užití navrhovaného systému

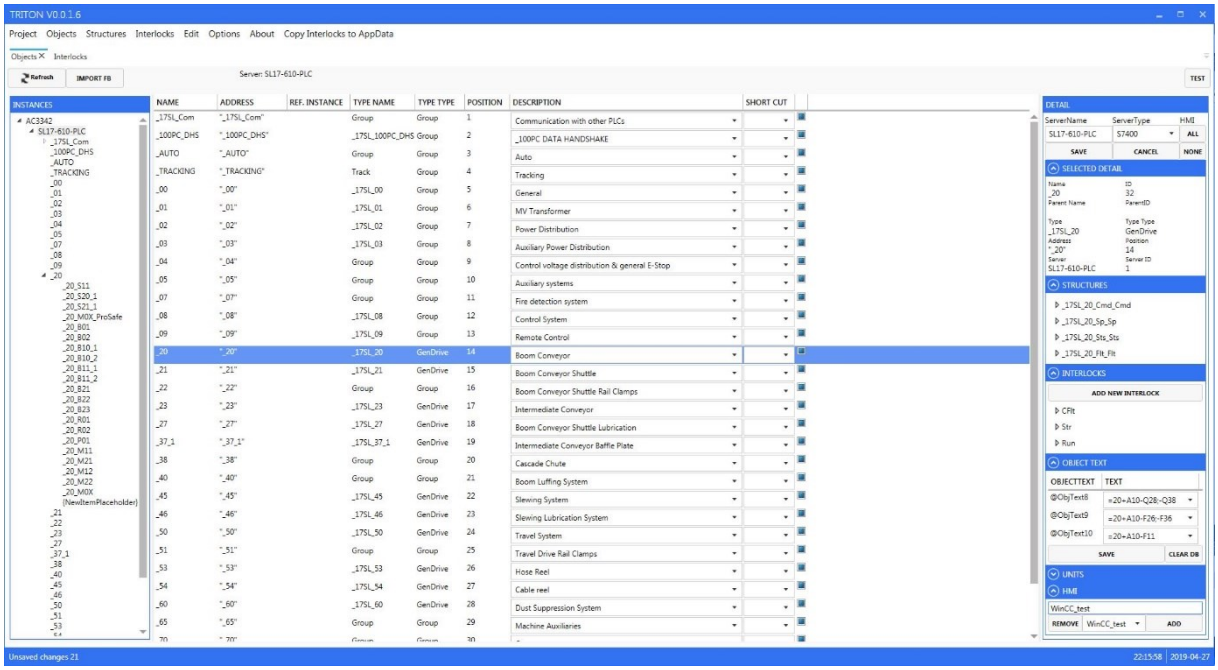
Při spuštění aplikace je prvním krokem připojení ke konkrétní databázi skrze úvodní okno pro vyplnění přihlašovacích údajů. Pole pro konfiguraci připojení jsou předvyplněná posledními platnými

údaji, pokud existují. Ty jsou uloženy spolu s dalšími údaji v souboru s uživatelskými daty s názvem *Ingeteam.Triton.Client.xml*. Tento soubor je vytvořen nebo modifikován při ukončování aplikace a je umístěn v adresáři: C:\Users\{uživatel}\AppData\Roaming\Ingeteam\....Uživatel má dále možnost si konfiguraci připojení uložit v podobě XML souboru, který může být rovněž zvolen jako zdroj přihlašovacích údajů v podobě konfiguračního souboru. Při zakládání nového projektu využije uživatel okno, pro definování základních propozic nového projektu. Po vyplnění názvu projektu, použitých jazyků a definování jednotlivých PLC je na základě konfigurace migrována nová databáze s vyplněnými číselníky a definovanými položkami.

Základní okno obsahuje navigační lištu s jednotlivými volbami pro zobrazení jednotlivých sekcí pro vytváření definic a oken pro editaci dalších vlastností projektu (správa textů, užitých vizualizací apod.). Hlavní okno je tvořeno na základě systému *AvalonDock*, kdy je jeho obsah tvořen záložkami s obsahem jednotlivých sekcí. Základními stavebními kameny jsou rozhraní pro definici objektů a rozhraní pro definování struktur.

Obě hlavní části mají obdobné základní uspořádání, na levé straně se nachází hierarchické zobrazení struktury v podobě *TreeView*. Pro záložku *Objects* je hlavním elementem hierarchie daný projekt, větvící se pak na užitá PLC a jejich příslušné objekty. V případě záložky *Struct* jsou výchozími položkami druhy Typu (Instanční, Skupiny, Sdílené apod.), jenž obsahují seznamy jednotlivých Typů a jejích Struktur proměnných. Prostřední část je vymezena již pro zobrazení příslušných položek vybraného objektu a samotné definování jejich vlastností. Některé vlastnosti jsou uživatelem vyplňovány ručně, jiné jsou vybrány pomocí prvku kombinovaného pole (combobox) z nabídky nebo jsou doplňovány automaticky.

Pravá část je věnována detailnějším informacím spojených s konkrétní položkou. Prvním krokem při vytváření definic technologických objektů je vytvoření Typů, kterých mohou objekty nabývat. Uživatel vytvoří novou instanci Typu pravým kliknutím na příslušný druh typu. Obdobně jsou vytvořeny struktury. Samotné proměnné jsou definovány a zobrazeny v prostřední části (název, výběr datového typu, jednotky, popis atd..). V levé části se nachází prostor pro editaci názvu a typu dané struktury a možnost uložení. Definování samotných objektů technologie probíhá obdobným způsobem. Obsah záložky pro definice objektů je znázorněn na Obr. 21. Po vytvoření instance PLC lze definovat jemu náležící objekty. Ve středové části opět definování základních vlastností a v pravé pak podrobnější informace specifikace vlastností vycházejících ze zvoleného Typu (jednotky a náhrady textu). Při vytváření jednotlivých definic je uživateli umožněno struktury a objekty kopírovat pro použití v jiných instancích. Dále je možné definované objekty a typy importovat do prostředí TIA Portal.



Obr. 21 Náhled uživatelského rozhraní pro definici objektů

Další část uživatelského rozhraní v záložce *Interlocks* je věnována komunikaci s rozhraním TIA Portal skrze nástroj TIA Openness. Uspořádání uživatelského panelu zobrazuje informace o hierarchické struktuře definovaných objektů podobně jako záložka *Objects*, avšak tato část není primárně určena k editacím. Horní část obsahuje panel se skupinou ovládacích prvků pro specifikaci generovaného výstupu. Specifikace spočívá v definování úložiště vytvořených souborů s podklady pro vizualizaci, jazyk překladu a varianta popisu. Panel je zobrazen na Obr. 22. Pro samotný export je nutné nejdříve připojit aplikaci k otevřenému projektu v TIA Portal. Dále pak v aplikaci Triton vybrat požadované objekty, pro které mají být na základě exportovaných dat do databáze vygenerovány interlocky a vytvořeny podklady pro vizualizaci. Popisu exportu dat z TIA Portal je věnována výše uvedená kapitola 4.2.1 Vytvořené obslužné třídy pro TIA Openness.

Obr. 22 Ovládací prvky pro generování interlocků a podkladů pro vizualizaci

5.4 Mechanismy zpracovávající vstupní data

Tato podkapitola bude věnována popisu vybraných procesů, které transformují data do vhodné formy pro příslušnou část procesu. Jedná se o úkony spojené s jejich zpracováním, prezentací uživateli, modifikace a převod do podoby určené pro uchování nesené informace v databázi.

- **Ukládání a mazání**

Ukládání změn do databáze může probíhat několika způsoby. Uživatel může během editace ukládat dílčí části definic objektů zvlášť (např. definování textových náhrad a jednotek u instancí objektů) nebo uložit daný objekt jako celek včetně jeho vnořených objektů. Lze tak ukládat jednotlivé technologické objekty PLC anebo celý projekt. Definice struktur lze ukládat obdobně ukládat jednotlivě nebo jako celé typy. Pro optimalizaci procesu ukládání větších celků je implementován systém pro sledování změn objektů. Při samotné transakci na databázi jsou ukládány pouze objekty, které byly skutečně pozměněny.

Již mnohokrát bylo uvedeno, že definice technologického objektu jako takového je v systému tvořena propojením několika entit napříč databázi. Veškeré operace týkající se databázových dat (CRUD) tak musí být prováděny v odpovídajícím pořadí, které respektuje vzájemné reference mezi databázovými objekty. Z tohoto pohledu je nejsložitější operace mazání a vytváření.

Před smazáním samotného objektu je potřeba vyhledat a ošetřit veškeré reference daného objektu. Například smazání některého z technologických objektů tak obnáší odstranění objektů definujících složený popis, náhrady textů, jednoty, přiřazené vizualizace a interloky včetně vygenerovaných položek a tak podobně.

Vazby mezi jednotlivými entitami jsou tvořeny pomocí cizích klíčů, kterými jsou odkazy na jednotlivá *Id* konkrétní entity. Tato *Id* jsou položkám přiřazována automaticky databázovým systémem až při samotném uložení záznamu. Aby bylo možné při vytváření jednotlivých částí definic vystavovat strukturu vzájemných vazeb, byl vytvořen systém pomocných identifikátorů. Ten zajišťuje korektní provázanost dat ještě před samotným uložením jednotlivých částí. Tento systém přiřazuje objektům při jejich vytvoření záporné identifikátory. Jedná se o sadu lokálních proměnných, které jsou po přiřazení automaticky dekrementovány a připraveny pro další objekty. Při ukládání jednotlivých záznamů je ze serverové části aplikace navráceno již skutečné identifikační číslo položky. V objektech, které mají vazbu na tuto položku, je pak původní cizí klíč v podobě lokálního identifikátoru nahrazen odpovídajícím údajem.

Některé z doménových objektů ve svých vlastnostech obsahují kromě reference v podobě cizího klíče i instanci tohoto referencovaného objektu. Důvodem je provázanost dat, která vytvářejí samotnou definici technologického objektu. Proto vzniká potřeba dostupnosti podrobnějších informací o instanci vnořeného objektu. Aby byla zajištěna provázanost a aktuálnost všech dat objektu, byly implementovány mechanismy, které při změně cizího klíče vyhledají v systému příslušný objekt a přiřadí jej danému objektu. Systém pracuje i v opačném směru. V případě, že je objektu přiřazen jiný referencovaný objekt, je změněn i cizí klíč.

- **Lazy load**

Veškeré operace týkající se modifikace dat jsou vykonávány na lokálních datech, která jsou do systému z databáze načtena. Objem přenášených dat z databáze je optimalizován pomocí postupného vyčítání dat až v případě, kdy jsou vyžadována. Princip systému *Lazy Load* již byl nastíněn v kapitole 5.2.1,věnující se popisu doménových objektů. Tento přístup přináší také časové zefektivnění

jednotlivých transakcí. Obsah metody *Load(bool aReload =false)*, kterou obsahují objekty využívající postupné načítání je dán vlastnostmi objektu. Vykonáním metody dojde k naplnění vlastností, které souvisejí s referencemi na jiné objekty. Jako příklad lze uvést objekt *Instace* v těle metody jsou postupně volány funkce *LoadInstances(aReload)* pro načtení seznamu vnořených technologických objektů, *LoadInterlockList(aReload)* pro načtení seznamu objektů interlokových podmínek, *LoadDescription(aReload)* pro objekty definující textový popis, *LoadObjectTextList(aReload)* pro definované náhrady textu, *LoadInstanceUnitList(aReload)* pro specifické jednotky objektu a *LoadVisualizationList(aReload)* pro seznam vizualizací se kterými je objekt svázán. Parametr *aReaload* je společný všem metodám a jedná se o informaci pro systém spravující lokální uložiště dat, který je popsán v předcházející kapitole 5.2.2 Systém lokálních dat a jejich verzování. Tyto metody lze využít i samostatně pro aktualizaci konkrétních dat.

- **Automatizované procesy uživatelského rozhraní**

Uživatelské rozhraní napomáhá uživateli v orientaci ve struktuře objektů a usnadňuje dodržování pravidel při jejich definování. Pro data zadávána uživatelem do jednotlivých částí uživatelského rozhraní jsou průběžně prováděny kontroly (vyplnění povinných položek, unikátnost názvů apod.). K těmto kontrolám a jejich indikaci je využito rozhraní *IDataErrorInfo* V případě chyby je uživatel informován o příčině chyby v místě výskytu chybně zadané hodnoty. Dále jsou v takovém případě znemožněny některé operace, jako například uložení.

Jedním z usnadnění pro zadávání dat je filtrace příslušných nabídek. Jedná se o datové typy, kterých mohou nabývat proměnné ve strukturách, druhy struktur, které mohou obsahovat definované typy a možnosti vkládat vnořené objekty.

Filtrování datových typu proměnných vychází z druhu struktury. Jedná-li se o některou strukturu ze základních typů struktur (*Sts,Sp,Cmd,Flt*) mohou obsahovat proměnné základních datových typů i vnořené substruktury. V takovém případě je datovým typem proměnné konkrétní struktura a tvoří tak odkaz na skupinu příslušných proměnných. Struktury typu *Share* (pro sdílené proměnné) a *Com* (pro proměnné využívané pro komunikaci) mohou obsahovat rovněž proměnné základních datových typů nebo vnořené speciální substruktury. Uvnitř struktur proměnných, které jsou definovány jako *SubStructure* se mohou nacházet proměnné pouze základních datových typů.

Omezení pro druhy struktur obsažených v typech je dáno z druhem typu. Typy patřící do skupiny *Instance, Group*,a *GenDrive* mohou zpravidla obsahovat až čtyři struktury kategorie *Sts,Sp,Cmd,Flt.* Druhy typů označovány jako *ShareDb* a *ComDb* mohou nabývat pouze jedné struktury proměnných z kategorií *Share* a *Com.*

Na základě zvolených datových typů proměnných jsou automaticky dopočítávány jejich relativní adresy v rámci struktury. Protože každý datový typ má svou charakteristickou velikost (*bool* – 1 bit, *byte* – 8 bitů, *float* – 32 bitů, apod…) je potřeba přepočítat na jaké pozici bude začínat další proměnná. Adresa odkazuje na konkrétní bit, kde vymezený prostor začíná. Formát adresy je tedy složen z čísla daného byte a příslušného bitu: *byte_bit*. Adresa následující proměnné pak začíná vždy na dalším byte. Výjimku tvoří jednobitové proměnné typu *bool*, které mohou pokračovat na stejném byte. V případě změny datového typu některé z proměnných nebo při změně jejich pořadí ve struktuře je nutné všechny adresy přepočítat. Příklad pro pět vzorových proměnných je uveden na Obr. 23 Jedná se o kombinaci čtyř proměnných základních datových typů a jedné položky představující substrukturu. V tomto případě se jedná o strukturu s osmi proměnnými typu *bool*, která má tedy velikost 1 byte. Obrázek zachycuje dvě varianty přiřazených adres v závislosti na uspořádání proměnných.

NAME	ADDRESS	POSITION	DATA TYPE
Item 1	0_0	1	bool
Item 2	0_1	2	bool
Item 3	1_0	3	GenOnOff_Sub
Item 4	2_0	4	bool
Item 5	3_0	5	float

NAME	ADDRESS	POSITION	DATA TYPE
Item 5	0_0	1	float
Item 2	4_0	2	bool
Item 4	4_1	3	bool
Item 1	4_2	4	bool
Item 3	5_0	5	GenOnOff_Sub

Obr. 23 Příklad přepočtu adres proměnných při změně pořadí

- **Textové popisy objektů**

Zvláštní pozornost si zaslouží také skládání textových popisů. Každý objekt technologie je identifikován svým názvem. Z názvu však nemusí být na první pohled zřejmé, jaký konkrétní prvek daný objekt představuje. Z tohoto důvodu obsahuje definice objektu textový popis, jenž poskytuje podrobnější informaci o jeho charakteru. Pokud se jedná o objekt typu skupiny, která sdružuje další vnořené objekty, pak vnořené objekty přebírají popis svého nadřazeného objektu, který je rozšířen o vlastní text. Na schématu uvedeném na Obr. 24 je znázorněn princip předávání popisu. Výsledný popis je složen z jednotlivých částí, které představují entity databáze *InstanceDescription*, jenž obsahují odkaz na příslušný text a jeho pozici v celkovém popisu. Uživatel může využít texty, které jsou již v databázi použity nebo vytvořit vlastní, který je následně do databáze vložen. Pomocí samostatného okna pro správu textů může být seznam textů editován. Tento způsob skládání textů byl zvolen pro budoucí zefektivnění a unifikaci procesu vytváření cizojazyčných překladů, neboť jsou eliminovány duplicitní záznamy.

Obr. 24 Schéma vytváření popisu vnořených objektů

Zdrojový kód na Obr. 25 obsluhuje proces sestavení výsledného popisu objektu. Jedná se o metodu náležící třídě *InstanceObject.cs*, která představuje technologické objekty. Prvním krokem je ověření, zda je objekt plně načten z databáze. Pokud ne, jsou z databáze vyčteny příslušné položky *InstanceDescription* pro sestavení popisu. Následuje sestavení popisu z částí, které jsou seřízeny podle pořadí a dohledání příslušných textových záznamů. Položka *LastDescription* představuje část popisu, která definuje daný objekt. Pokud popis není definován, je použit jako indikace uživateli výchozí řetězec: `-new text-`. Předcházející části popisu vycházejí z nadřazeného objektu, kterému tento objekt náleží. Posledním krokem je předání současného stavu vnořeným objektům. Při změně popisu některé z nadřazených položek jsou tak změny promítnuty i na popisy jejich referencovaných objektů. Pro popis

daného objektu je možné definovat zkrácený popis. Jedná se o náhradu popisu, která může výt využita při sestavování popisu vnořených objektů.

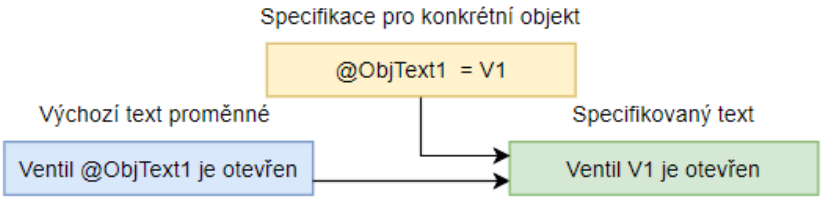
```
public void SetDescriptionString(object aSender = null, ChangeIndicatorEventArgs aArgs = null)
{
    if (!IsLoaded)
        LoadDescription();
    var SortedByPosition = InstanceDescriptionList.OrderBy(x => x.Position).ToList();
    var lDescriptionString = "";
    foreach (InstanceDescriptionObject Description in SortedByPosition)
    {
        string lText="-new text-";
        if(Description.MultilingualStringId !=null)
            lText = DataStoreTypes.I.GetMultilinguaStringlList().FirstOrDefault(a => a.Id == Description.MultilingualStringId).Text;
        lDescriptionString += lText + " ";
    }
    DescriptionString = lDescriptionString;
    if (SortedByPosition.Count>0)
        LastDescription = SortedByPosition.Last();

    if (aSender!=null && aSender is InstanceDescriptionObject)
        SetChangeToReferenced(aSender as InstanceDescriptionObject);
    ////// SHORTCUT
    SetShortDescription();
}
```

Obr. 25 Tělo metody pro sestavení popisu objektu

- **Specifikace proměnných**

V předcházejících kapitolách již bylo mnohokrát zmíněno, že struktury proměnných jsou přiřazeny danému Typu, který může tvořit předpis pro několik různých objektů. Stejně jako objekt, může mít proměnná přiřazen popis. Pro jednotlivé tagy proměnných jsou pomocí textových popisů proměnných generovány podklady pro vizualizaci. V některých případech je nutné v obecném textu specifikovat, kterému objektu se konkrétní proměnná váže. K tomuto účelu slouží nahraditelná část popisu tzv. *ObjectText*. Skrze klíčové řetězce *@ObjText1, @ObjText2, ..., je* možné pro každý objekt definovat specifické náhrady za jednotlivé elementy. Seznam náhrad je definován na základě typu, který je objektu přiřazen, a uživatelem zvolenými texty. Schéma principu náhrad textů je uvedeno na Obr. 26.



Obr. 26 Schéma principu náhrad textů

Na podobném principu lze definovat i specifické jednotky proměnných. Pokud technické řešení umožňuje, aby některá proměnná prezentovala hodnoty různých jednotek, lze je specifikovat pro konkrétní objekt. V takovém případě je ze seznamu jednotek proměnné přiřazen typ, který označuje, že se jedná o specifickou jednotku. Na základě typu objektu je pak pro tento objekt vytvořen seznam specifických jednotek. Skutečné jednotky vybraných proměnných pak navolí uživatel pro konkrétní objekt. V databázi jsou tato data uchovávána pomocí entit *InstanceUnit*

- **Generované části**

Prvním procesem generování dat je již zmiňované vkládání položek interlocků do databáze. Proces exportu dat z TIA portálu ve formě dočasných XML souborů a jejich deserializace je nastíněn v kapitole 4.2 TIA Portal Openness. Z instance odpovídající příslušnému XML souboru, resp. funkčnímu bloku, jsou extrahovány logické vazby mezi proměnnými. K tomuto konkrétnímu kroku byla využita komponenta vytvořena firmou Ingetem a.s. Části kódů v TIA Portal (network), které definují interlokové podmínky musí být v komentáři označeny identifikátorem @IL nebo @Cflt. Výstupem jsou objekty interlocku, včetně jeho položek, pro příslušnou instanci. Ty jsou následně uloženy do databáze jako entity tabulek *Interlocks* a *InterlocksItem*. Položky interlocku představují tagy jednotlivých proměnných, ke kterým se interlock vztahuje. Pokud funkční bloky obsahují složitější logiku, je do jedné položky slučováno více tagu. Na Obr. 27 je uveden náhled na části generovaných položek databáze. V uvedeném příkladu se jedná o dva interlocky, pro spuštění a vypnutí zařízení.

Id	Name	InstanceId
118	CtrlOn	7
119	CtrlOff	7

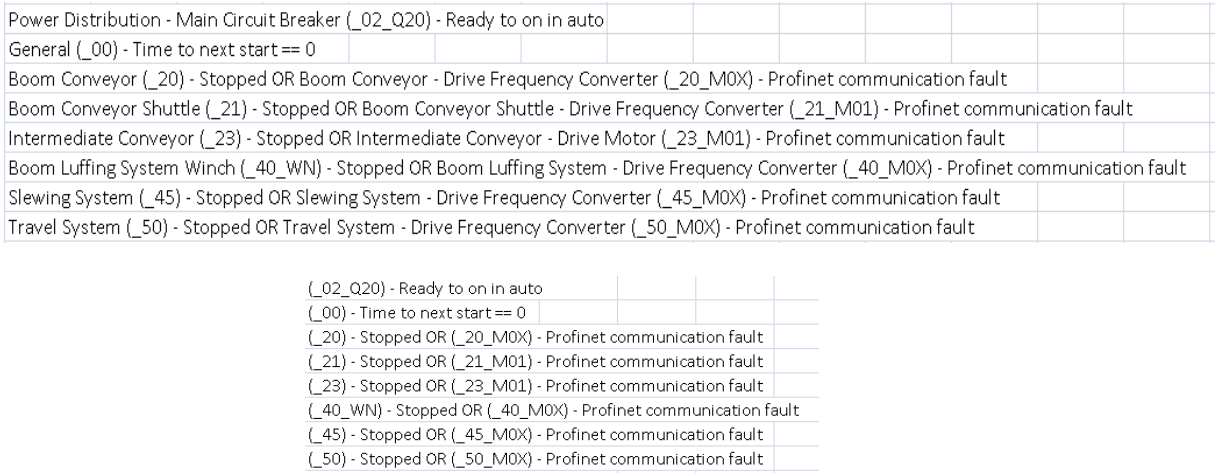
Id	Bit	InLckCode	InterlockId
499	0	"_02_Q20".Sts.RdyToOn	118
500	1	("_00".Sts.TmNextStr) == (_00.0)	118
501	0	("_20".Sts.Stpd) OR ("_20_M0X".Sts.CmnFlt)	119
502	1	("_21".Sts.Stpd) OR ("_21_M01".Sts.CmnFlt)	119
503	2	("_23".Sts.Stpd) OR ("_23_M01".Sts.CmnFlt)	119
504	3	("_40_WN".Sts.Stpd) OR ("_40_M0X".Sts.CmnFlt)	119
505	4	("_45".Sts.Stpd) OR ("_45_M0X".Sts.CmnFlt)	119
506	5	("_50".Sts.Stpd) OR ("_50_M0X".Sts.CmnFlt)	119

Obr. 27 Náhled na vybrané sloupce záznamů databáze pro entity Interlocks (nahoře) a InterlocksItem (dole)

Kromě s interlocků a jejich položek, jsou do databáze vloženy i textové popisy lokálních proměnných, které jsou rovněž dostupné v exportovaných XML souborech.

Dalším procesem automatického generování, které aplikace Triton poskytuje je vytváření podkladů pro vizualizace. Jedním z předmětů této diplomové práce bylo právě generování odpovídajících textových popisů interlockových položek pro tyto podklady. Jak již bylo zmíněno, tyto položky představují odkazy na konkrétní proměnné dané struktury, která je dána zvoleným Typem objektu. Výsledný popis je složen z popisu daného objektu a popisu dané proměnné. Pokud se jedná o interlockovou položku složenou z více tagů, je prvním krokem, před zahájením generování textu, oddělení jednotlivých části a získání logických operandů. Výsledný textový řetězec je pak postupně vystavován z vygenerovaných textových popisů jednotlivých tagů a vkládaných operandů. Proces generování popisu obstarává vytvořená metoda *GetDescriptionFromCode(string aInlcCode, InstanceDetail aInstance, Languages aSelectedLanguage, int aIndex=0)*. Generování začíná rozdělením tagu na jednotlivé části. Jedná se tedy o elementy identifikující objekt, strukturu a proměnnou. Speciálním případem jsou tagy pro lokální proměnné a tagy odkazující se na konstantu. V těchto případech se nevyskytuje element pro identifikaci struktury. Následně jsou pro jednotlivé části získány příslušné texty. Výchozím bodem je vyhledání příslušného technologického objektu a jeho popisu v databázi. Poté je ve strukturách příslušného typu vyhledána konkrétní proměnná a v databázi je dohledán příslušný popis. V případě, že jsou pro objekt definovány náhrady textu, dojde k jejich

substituci a je navrácen popis s již specifikovaný popis proměnné. Pokud tag obsahuje symbol pro negaci je do popisu proměnné vložen příslušný řetězec (např, NOT pro anglickou variantu textu). V základní variantě je výsledný popis tagu složen v podobě: *Popis objektu (Název Objektu) – Popis proměnné.* Pro zkrácenou verzi je text neobsahuje popis objekt, ale jen jeho název. Na uvedeném Obr. 28 je znázorněn výpis generovaných textů, které odpovídají položkám na Obr. 27.



Obr. 28 Ukázky vygenerovaných popisů v plné variantě (nahoře) a zkrácené (dole)

Generování podkladů je možné také v jiném jazyce, než je základní jazyk, ve kterém jsou texty uloženy v databázi (tabulka *MultilingualString*). K tomuto účelu slouží systém překladů. V případě generování v jiném jazyce jsou pro příslušné texty již při generování popisu vyhledávány překlady. Výsledný řetězec je sestaven již z přeložených textů.

Závěr

Cílem této diplomové práce byl návrh a zhotovení systému umožňující vytváření definic technologických objektů využívaných pro tvorbu řídicích programů automatizovaných systému. Tyto definice umožnují programátorům dělení objektů do dílčích kategorií a skupin, dále pak specifikaci vlastností objektů skrze definované typy. Na základě těchto definic je vytvářen řídicí program technologie. Práce byla vytvořena ve spolupráci s firmou Ingeteam a.s. Struktura jednotlivých částí práce je proto vytvořena tak, aby zapadala do reálného procesu vývoje řídicích technologií využívaného firmou.

Náplň práce je složena ze tří pomyslných částí, které spolu úzce souvisí a společně tak tvoří výsledný systém. První část představuje návrh a vytvoření SQL databáze, jejíž struktura odráží současnou skladbu definičních listin v podobě rozsáhlých XLSM souborů. Dále implementace uživatelského rozhraní pro správu databázových dat pro vytváření typů, které sdružují struktury proměnných a definice samotných objektů. Obslužný software, nesoucí název Triton, umožnuje hierarchický přehled objektů a jejich vlastností, automatické kontroly vstupních dat, filtrace voleb a poskytuje možnost tvorby automaticky generovaných dat. Součástí aplikace je možnost komunikace s TIA Portalem pomocí nástroje Openness. Ten umožnuje přenos dat mezi databází a programovacím rozhraním skrze importní a exportní XML soubory. Díky této komunikaci lze vkládat základní definice objektů v podobě funkčních bloků do TIA Portalu. Po vytvoření logických vazeb mezi objekty, resp. odpovídajících funkčních bloků, a jejich proměnnými jsou pomocí aplikace Triton na základě interlokových podmínek generovány podklady pro vizualizaci.

Vypracování této diplomové práce vyžadovalo nastudování poměrně širokého okruhu témat. Jedná se o problematiku z oblasti vývoje řídicích programů pro PCL a firemního standardu pro tento proces. Dále bylo nutné nastudovat problematiku databázových systému a jejich implementace, návrhových vzorů a postupů pro efektivní práci s daty.

Již během vývoje celého systému vznikla potřeba některé jeho části již reálně využívat při procesu vývoje řídicích programu technologie. Z tohoto důvodu byly upřednostňovány práce na částech týkajících se generovaných dat a komunikace s TIA Portalem. Proto byla po navržení databázového systému vytvořena základní kostra aplikace pro práci s databázovými daty využívající Entity Framework. Jako další krok byly ze zmiňovaných důvodů implementovány elementární metody pro práci s TIA Poratelm. Základní části databáze byly napleněny reálnými daty z definičních listin ve formě XLSM souboru. K tomuto kroku transformace dat z XLSM souborů do prázdné databáze, která obsahovala pouze číselníky s výchozími daty, bylo využito separátního softwarového nástroje vytvořeného firmou Ingeteam a.s. Na těchto reálných datech pak bylo odladěno generování interlockových objektů z logických vazeb uvnitř funkčních bloků a jejich vkládání do databáze. Dále byla vystavěna primární verze metod pro generování textových popisů interlockových položek v základním jazyce.

Následně byly zdokonalovány oddíly aplikace pro prezentaci dat uživateli a vyvíjeny části uživatelského rozhraní pro samotné vytváření definic objektů a jejich modifikace. Paralelně s těmito kroky byly postupně zdokonalovány procesy pro práci s TIA Portalem a rozšířeny možnosti pro generované texty. Těmito rozšířeními jsou například možnosti volby jazyka překladu a variant formátování textu.

Možnost naplnění databáze reálnými daty usnadnila další vývoj systému v oblasti týkající se práce s definicemi objektů. Přestože definování struktury databáze bylo prvním krokem při vývoji systému, bylo i posléze nutné provádět její modifikace. Jednalo se o změny zahrnující nově vzniklé skutečností během vývoje. Ke zpětné analýze chyb při vývoji a testování systému v reálném prostředí napomáhá systém logování jednotlivých procesů systému a chybových stavů. Velikost databáze naplněné reálnými daty odpovídá řádové desítkám MB, samozřejmě však záleží na rozsáhlosti celého projektu.

Byl vytvořen základní prvek pro proces tvorby obslužného programu pro řízenou technologii. Ten tvoří centrální bod, ve kterém jsou jednotlivé části vývoje procesovány, a poskytuje tak platformu pro unifikaci celého postupu. Navrhovaný systém a vytvořená aplikace, které jsou předmětem této diplomové práce, jsou proto vytvořeny tak, aby mohly být předmětem dalšího vývoje firmy Ingeteam a.s. Jedná se například o rozšiřování funkcionality o komponentu pro konverzi a přenos vybraných dat do další části spojené s řídicím programem technologie, kterou představuje databáze AppData s parametry řízeného procesu. Systém je koncipován tak aby umožňoval práci na lokální databázi, i na databázi jejíž umístění je mimo zařízení uživatele. Tento předpoklad umožnuje rozvoj myšlenky, že by si uživatel mohl kopírovat data uložená v databázi přístupné pouze z uzavřené firemní sítě na své lokální pracoviště. Mohl by tak pokračovat v práci i mimo dosah této sítě. Po opětovném připojení by vložil modifikovaná data do hlavní databáze. Tento koncept však vyžaduje implementaci rozšiřujícího systému pro verzování dat v samotné databázi, a v neposlední řadě také systém pro porovnávání a řešení konfliktních změn mezi více uživateli.

Použitá literatura

[1] BENEŠ, Pavel, et al. *Automatizace a automatizační technika. 1, Systémové pojetí automatizace.* Brno: Computer Press, 2012. ISBN 978-80-251-3628-7.

[2] CORONEL, Carlos and Steven MORRIS. *Database Systems: Design, Implementation, & Management*. 12 edition. Boston: Course Technology, 2016. ISBN 978-1305627482.

[3] DRISCOLL, Brian, Larry TENNY, Rob VETTOR a Nitin GUPTA. *Entity Framework 6 Recipes, Second Edition* [online]. Apress, 2013 [cit. 2019-03-04]. ISBN 978-1-4302-5789-9. Dostupné z: http://iliasoft.ir/Download/Apress_Entity_Framework_6_Recipes.pdf

[4] GAMMA, Erich. *Design patterns: elements of reusable object-oriented software*. Reading, Mass.: Addison-Wesley, c1995. ISBN 978-0201633610.

[5] LERMAN, Julia. *Programming Entity Framework, Second Edition* 2010 [cit. 2019-03-04]. ISBN 987-0-596-80726-9.

[6] PETZOLD, Charles. *Mistrovství ve Windows Presentation Foundation: aplikace = kód + markup*. Brno: Computer Press, 2008. Mistrovství. ISBN 978-80-251-2141-2.

[7] ZEZULKA, František. *Průmyslová automatizace: (teze přednášky k profesorskému jmenovacímu řízení).* Brno: VUTIUM, 2000. Vědecké spisy Vysokého učení technického v Brně. ISBN80-214-1634-3.

[8] BAŠINAR, Václav. *Návrh datové vrstvy pro business aplikaci v n-vrstvé architektuře s využitím technologie Microsoft Entity Framework 4.0*. Ostrava, 2011. Diplomová práce. VŠB – Technická univerzita Ostrava.

[9] JANEČEK, Pavel. *AUTOMATICKÉ GENEROVÁNÍ PLC PROGRAMU POMOCÍ TIA PORTAL OPENNESS*. Brno, 2018. Bakalářská práce. Vysoké učení technické v Brně.

[10] KOUBA, Petr. *TECHNOLOGIE SILVERLIGHT*. Brno, 2011. Diplomová práce. Vysoké učení technické v Brně.

[11] STRAKA, Jiří. *Převod Windows Forms do Windows Presentation Foundation*. Praha, 2011. Bakalářská. Unicorn College.

[12] DAJBYCH, Václav. *Model-view-viewmodel*. *DOTNETPORTAL.cz* [online]. 21.4.2009 [cit. 2019-03-04]. Dostupné z: https://www.dotnetportal.cz/clanek/4994/MVVM-Model-View-ViewModel

[13] HERCEG, Tomáš. Zajímavé návrhové vzory v praxi. In: *Wug.cz* [online]. 2015 [cit. 2019-03-04]. Dostupné z: https://wug.cz/zaznamy/301-Zajimave-navrhove-vzory-v-praxi

[14] KOČÍ, Michal. Co je XML?. *Interval.cz* [online]. 21.2.2000 [cit. 2019-03-04]. Dostupné z: https://www.interval.cz/clanky/co-je-xml/

[15] ROBERTSON, James a Suzane ROBERTSON. *Requirements Specification Template*[online]. Atlantic Systems Guild. 2003 [cit. 2019-03-04]. Dostupné z: https://www.volere.org/

[16] VALÁŠEK, Michal. *Extension methods, lambda expressions a LINQ v C#* In: Youtube.com [online]. 03.03.2007 [cit. 2019-03-04]. Dostupné z: https://www.youtube.com/watch?v=BXIrnD6DygU , Kanál uživatele: Altairis

[17] VALÁŠEK, Michal. *Jemný úvod do ORM a Entity Frameworku* In: Youtube.com [online]. 18.03.2007 [cit. 2019-03-04]. Dostupné z: https://www.youtube.com/watch?v=0nM38vBk6LI , Kanál uživatele: Altairis

[18] *Docs.microsoft.com: ADO.NET*[online]. 2017 [cit. 2019-03-04]. Dostupné z: https://docs.microsoft.com/cs-cz/dotnet/framework/data/adonet/

[19] *EntityFramework.net: Supported Database Providers* [online]. [cit. 2019-03-04]. Dostupné z: https://entityframework.net/supported-database-providers

[20] *Popis standardu programování aplikačního SW v prostředí Step7 , ppt*. Ingeteam a.s, Ostrava, 2016.

[21] *Automating projects with scripts, System Manual, 12/2017, Printout of the online help.* SIEMENS 2017

[22] *TIA Portal Openness: Introduction and Demo Application* [online]. SIEMENS, 2018 [cit. 2019-03-04]. Dostupné z: https://support.industry.siemens.com/cs/document/108716692/tia-portal-openness%3A-introduction-and-demo-application?dti=0&lc=en-WW

[23] *Programové bloky a editor S7-1200* [online]. Simexcontrol, 2007 [cit. 2019-03-04]. Dostupné z: https://w5.siemens.com/web/cz/cz/corporate/portal/home/produkty_a_sluzby/IADT/tia_na_dosah/Documents/2014_letni_akademie/3_Programovani.ppt

[24] *Programming Guideline for S7-1200/1500: STEP 7 (TIA Portal)* [online]. SIEMENS, 2014 [cit. 2019-03-04]. Dostupné z: https://www.industry.siemens.nl/automation/nl/nl/industriele-automatisering/industrial-automation/simatic-controller/modulaire-controllers/simatic-s7-1500/Documents/81318674_Programming_guideline_DOKU_v12_en.pdf

[25] *TIA Portal Openness – UsageFile / EnablerFile* [online]. SIEMENS, 2014 [cit. 2019-03-04]. Dostupné z: https://cache.industry.siemens.com/dl/files/307/103627307/att_38599/v2/tia_portal_openness_guidance_usagefile_enablerfile_v13sp1.pdf

Seznam příloh

I. Obsah přiloženého DVD

- Vytvořená aplikace.
- Části zdrojových kódů aplikace.
- Vzorový XML soubor vygenerovaný pomocí TIA Portal Openness.

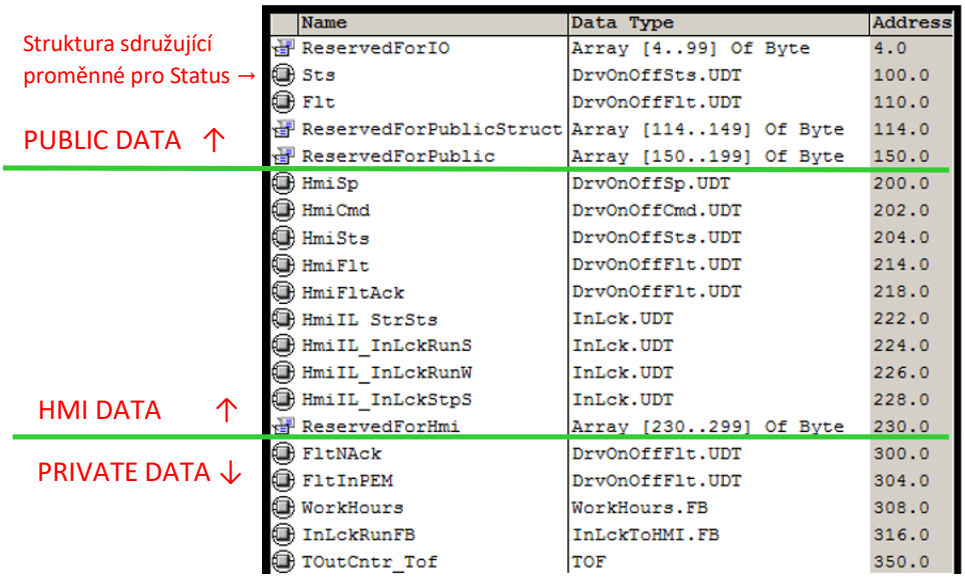
II. Vnitřní uspořádání datového bloku

V této přiložené kapitole je popsán systém uspořádání dat uvnitř instančního datového bloku v prostředí Step 7. Uspořádání lze rozdělit do tří hlavních částí.

První částí jsou takzvaná *Public data*. Tato data začínají na adrese 100.0. Prostor před je vyhrazen pro vstupní a výstupní proměnné daného bloku. Public data jsou vymezeným prostorem pro data představující veřejné proměnné a signály, které mohou být čteny jinými funkčními bloky. Protože na tyto proměnné mohou být navázány jiné části programu, je nutné, aby proměnné neměnily svůj význam ani adresu v bloku. Změny provedené mimo veřejnou část tyto proměnné neovlivňují – mají vliv pouze na jejich výsledné hodnoty. Proto změny programu provedené uvnitř příslušného funkčního bloku nemají vliv na zbytek vnějšího závislého kódu. V případě nutnosti změn ve veřejné části, je potřeba zajistit aby byly změny zohledněny ve všech částech programu, které s těmito daty pracují. Vývoj programu využívá objektově orientovaný přístup. Z tohoto důvodu se v této části bloku nachází struktury proměnných, které popisují a charakterizují objekt příslušného datového bloku jako celku, bez návaznosti na vnitřní stavy či proměnné. Těchto struktur tak mohou využít bloky, které jsou navzájem funkčně provázány a vyžadují informace o stavu jiného objektu. V této části datového bloku se proto nachází jedna z nejdůležitějších struktur *Status*, která je svými proměnnými nositelem informace o aktuálních stavech objektu. Další neméně významnými strukturami jsou například struktury *Fault*. Dle potřeby zde mohou být umístěny i další veřejné struktury či proměnné, pro které je rezervován příslušný prostor v paměti. Tyto rezervy na konci každého pomyslného oddílu zajišťují zachování adres struktur v případě přidání nebo odebrání nějaké proměnné v předcházejícím oddílu. Adresy po sobě následujících struktur jsou dopočítávány podle velikosti předcházející struktury.

Následujícím úsekem proměnných jsou data určená pro rozhraní operátorské stanice. Oblast dat začíná zpravidla na adrese 200.0. Předcházející pozice jsou rezervou předcházející oblasti pro public data. Protože se jedná o struktury proměnných, které jsou využívány HMI, mohou být jejich hodnoty nejen vyčítány ale také přepisovány. Aby byla oddělena programová a vizualizační část řídicí technologie, jsou některé struktury pro HMI kopiemi původních struktur. Příkladem jsou struktury *HMISts* a *HMIFlt*, které odpovídají kopiím struktur Status (Sts) a Fault (Flt). Hlavním z důvodu pro toto oddělení je možnost paralelního vývoje jak řídicího programu, tak vizualizace. Pokud by struktury nebyly duplikovány, spuštění simulace na vizualizaci by způsobovalo přepisování dat ve struktuře Status a tím by ovlivňovala vyvíjený program. Stejně jako oblast pro public data i oblast pro HMI rozhraní obsahuje datovou rezervu.

Zbylý prostor instančního datového bloku náleží strukturám proměnných, se kterými se pracuje pouze v rámci příslušného funkčního bloku. Tato oblast je označována jako *Private data.* Změny proměnných se projeví ve všech instancích změněného funkčního bloku, ale nijak neovlivní jiné části programu. Jedná se o lokální proměnné, které nejsou využívány v jiných blocích. Data těchto proměnných jsou uložena za rezervovaným prostorem pro HMI a začínají tak z pravidla na adrese 300.0.



Name	Data Type	Address
ReservedForIO	Array [4..99] Of Byte	4.0
Sts	DrvOnOffSts.UDT	100.0
Flt	DrvOnOffFlt.UDT	110.0
ReservedForPublicStruct	Array [114..149] Of Byte	114.0
ReservedForPublic	Array [150..199] Of Byte	150.0
HmiSp	DrvOnOffSp.UDT	200.0
HmiCmd	DrvOnOffCmd.UDT	202.0
HmiSts	DrvOnOffSts.UDT	204.0
HmiFlt	DrvOnOffFlt.UDT	214.0
HmiFltAck	DrvOnOffFlt.UDT	218.0
HmiIL_StrSts	InLck.UDT	222.0
HmiIL_InLckRunS	InLck.UDT	224.0
HmiIL_InLckRunW	InLck.UDT	226.0
HmiIL_InLckStpS	InLck.UDT	228.0
ReservedForHmi	Array [230..299] Of Byte	230.0
FltNAck	DrvOnOffFlt.UDT	300.0
FltInPEM	DrvOnOffFlt.UDT	304.0
WorkHours	WorkHours.FB	308.0
InLckRunFB	InLckToHMI.FB	316.0
TOutCntr_Tof	TOF	350.0

II.a Struktura dat v instančním datovém bloku [20]

Stejně jako základní objekty i některé skupiny (*Group* a *GenDrive*) mohou využívat společný interface proměnných. Záleží však na konkrétní situaci a úsudku programátora, zda je výhodné využít společného předpisu nebo je pro danou skupinu vhodnější vytvořit samostatný předpis. Struktura datového bloku skupin se liší. Není zde oblast pro vstupní a výstupní parametry a oblast pro veřejné struktury proměnných a proměnné začíná na adrese 0.0. Vymezená oblast pro HMI rozhraní začíná na adrese 1000. Vnitřní data začínají od adresy 2000.0 nebo 3000.0, v závislosti na rozsáhlosti skupiny. Ve veřejné části skupin se z pravidla nachází i struktura *Commnad*, což umožnuje, aby i pro vnitřní bloky skupiny byly dostupné informace o povelech pro danou skupinu. V praxi je toho využito například v případě, že pokud přijde na skupinu povel pro potvrzení procesu, jsou vnitřní objekty schopny tento povel převzít.

III. Doplňující informace k datovým vazbám mezi vrstvami MVVM modelu

Mezi nejvyužívanější parametry pro specifikaci datových vazeb pro tzv. *Binding* patří:

- **Mode**

Tento parametr definuje režim, určující jakým způsobem má být zdroj a cílová vlastnost svázána. Typ zvolené vazby závisí na charakteru dat. Rozlišujeme tři možné režimy. Prvním je *OneTime*, kdy k odeslání dat do cílového prvku dojde pouze jednou a to v okamžiku, kdy je zdrojová hodnota poprvé nastavena. Pokud je při běhu programu tato hodnota změněna, uživatelské rozhraní není o této události informováno a změna se neprojeví. Tento režim je vhodný pro data, která jsou určena pouze pro čtení a neočekávají se jejich další změny, případně není žádoucí, aby byly změny zaznamenány. Pro případ, kdy je potřeba zobrazovat aktualizovanou hodnotu slouží režim *OneWay*. Tento režim odesílá zdrojovou hodnotu do cíle při jejím nastavení a díky výše zmiňovaným notifikačním rozhraním informuje uživatelské rozhraní o každé změně své hodnoty. Tento režim neumožnuje uživateli měnit navázanou hodnotu, tuto funkci přináší až vazba *TwoWay*, která umožnuje obousměrnou vazbu mezi zdrojem a cílem. Pokud není mód režimu nastaven, je užit defaultní nastavení, které nastaví režim na základě typu prvku. To znamená, že jedná-li se o pouze zobrazovací prvek, je mód defaultně nastaven do režimu *OneWay*, jedná-li se naopak o prvek umožňující editaci, je jako výchozí režim zvolen *TwoWay*.

- **UpdateSourceTrigger**

Nastavení tohoto parametru definuje, kdy bude uživatelské prostředí občerstvováno aktualizovanými daty. Obdobně jako v předchozím případě lze zvolit kromě defaultní varianty, která je zvolena podle předpokládaného chování daného prvku, ze tří různých režimů. Možnost *PropertyChange* udává, že bude hodnota aktualizována při každé změně zdrojové (nebo v případě *TwoWay* modu - editované) hodnoty. Další variantou je režim *LostFocus*, ten je vhodné využít v případě, že je z uživatelského rozhraní editován zdroj, při tomto nastavení bude změna hodnoty zdrojem převzata až po dokončení editace prvku a focus získá jiný prvek UI. Příkladem může být využití u textového pole, kdy nebude nedocházet k změně zdrojového textového řetězce s každým přidaným nebo změněným znakem, ale až v okamžiku kdy uživatel prvek textového opustí. Poslední možností je možnost Explicit, při jejímž použití je aktualizace hodnot řízena manuálně voláním metody *UpdateSource()*.

- **Converter**

Při nastavování datových vazeb je nutné dbát na dodržení datových typů vstupních hodnot. Některé vlastnosti prvků uživatelského rozhraní mohou být výčtového typu a tak je nelze přímo navázat na zdrojová data. Přítomnost typů, které náleží výlučně prezenční vrstvě, by mělo za následek nežádoucí provázání vrstev. Pro tyto případy lze využít právě tohoto elementu. Jedná se o pomocné třídy, jejichž úkolem je přetypovávání hodnot z typu zdroje na požadovaný typ cílové vlastnosti a naopak. Případně se může jednat o jinou speciální úpravu vstupní hodnoty. Systémový framework poskytuje několik základních konvertorů, ale v běžné praxi je potřeba implementovat konvertory pro různé specifické případy. Pro tvorbu takových tříd je důležité, aby implementovaly rozhraní *IValueConverter*, případně *IMultiValueConverter,*

- **Source**

Při nastavení tohoto parametru je potlačena vlastnost *DataContext* nadřazeného prvku, který vnořené prvky automaticky přebírají a pro daný prvek tak může být zdroj vazby nastaven na jiný objekt.

- **ValidatesOnNotifyDataErrors / ValidatesOnDataErrors**

Nastavením těchto booleovských parametrů je specifikováno vyvolávání událostí zajišťujících chování při chybném uživatelském vstupu. Tyto události pro validace vstupních dat jsou obsluhovány pomocí implementovaných rozhraní *IDataErrorInfo* a *INotifyDataErrorInfo*, která jsou součástí frameworku.

[6][10][12]

Níže je uveden jednoduchý příklad XAML definice prvku uživatelského rozhraní *TextBox* včetně specifikace datové vazby. Zápis deklaruje, že v textovém poli bude zobrazena vlastnost *Name*, která náleží objektu s názvem *SelectedTreeItem*. Tento objekt musí být součástí datového zdroje daného XAML dokumentu a musí mít naimplementováno notifikační rozhraní. Dále je dáno, že se jedná o dvousměrnou vazbu, která vyvolává požadavek na aktualizaci hodnot při každé změně. Hodnota podléhá validační kontrole, která bude vyvolávat s tím spojené události a pro vlastnost *Visibility* je užit *Converter* .

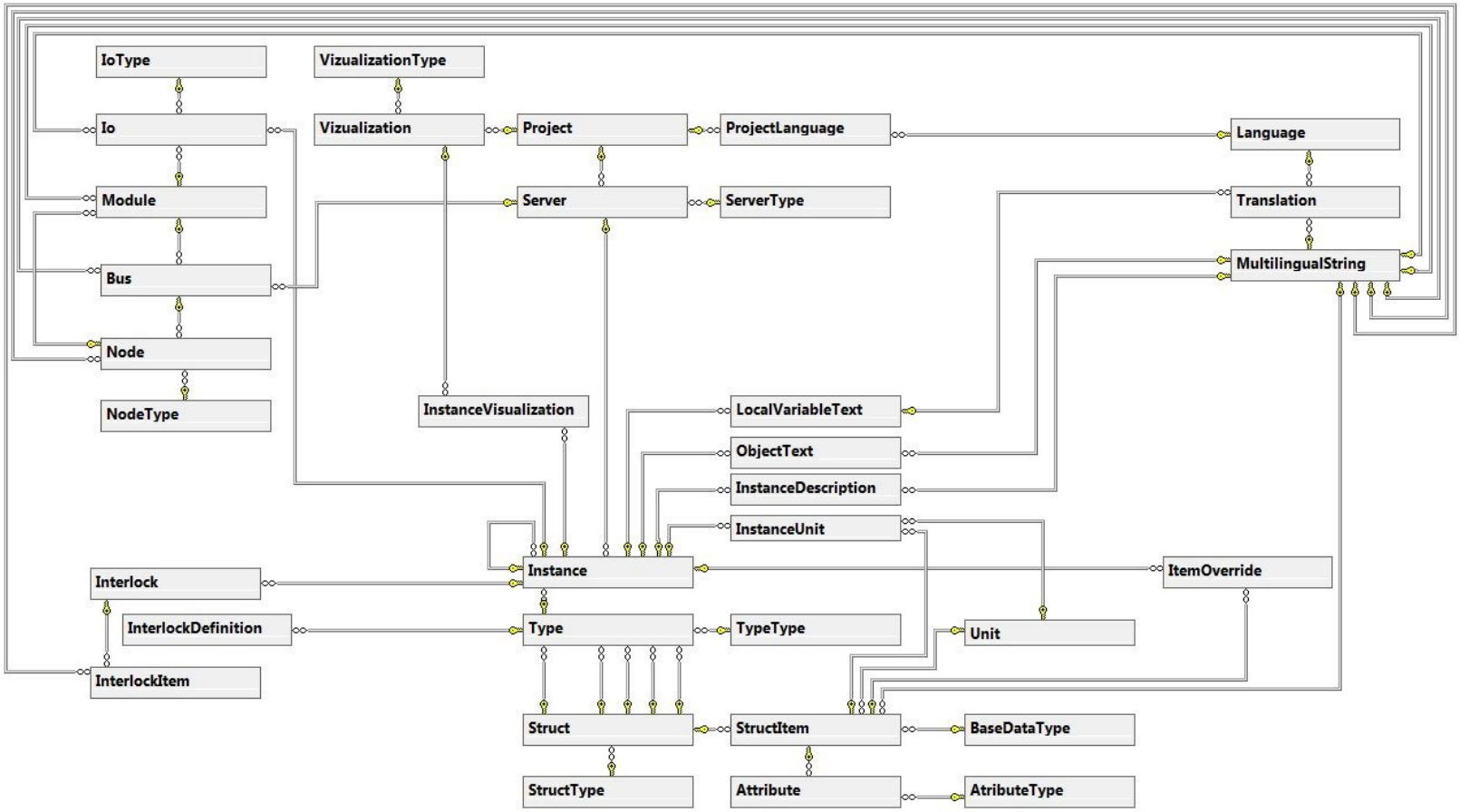
```
<TextBox Grid.Row="1" Grid.Column="0" Text="{Binding SelectedTreeItem.Name, Mode=TwoWay,
UpdateSourceTrigger=PropertyChanged, ValidatesOnDataErrors=True}" Visibility="{Binding IsVisible,
Converter=VisibilityConverter}" />
```

K obsluze interakcí uživatele s uživatelským rozhraním je využíván element *command*, jedná se o příkaz, který obstarávaná obsluhu vykonávání metody přiřazené příkazu. Pro objasnění principu využití tohoto elementu je níže uveden příklad.

Jedná se o definici tlačítka s přiřazeným příkazem *LoadSelectedCommand* který vyvolá příslušnou metodu po kliknutí na tlačítko. Dále je nastaven předávaný parametr, kterým je v tomto případě odkaz na prvek s názvem *MyListBox* a jeho vlastnost *SelectedItem*. Do parametru přiřazené metody je tak předána vybraná položka z daného listboxu. V těle definice prvku je skrze elementy *EventTrigger* a *InvokeCommandAction* vnořeno přiřazení commandu na libovolnou událost. V tomto případě se jedná o událost *MouseEnter*, která je vyvolána pokud se kurzor myši dostane do prostoru tlačítka.

```
<Button Command="{Binding LoadSelectedCommand}" CommandParameter="{Binding
ElementName=MyListBox, Path=SelectedItem}">
    <i:Interaction.Triggers>
        <i:EventTrigger EventName="MouseEnter">
            <i:InvokeCommandAction Command="{Binding MouseOnButtonCommand}"/>
        </i:EventTrigger>
    </i:Interaction.Triggers>
</Button>
```

IV. Diagram struktury databáze



IV a. Diagram struktury databáze pro definice technologických objektů